```
**********************************************************
   55541 Wed Nov 25 13:59:35 2015
new/usr/src/uts/common/exec/elf/elf.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1721 int
1722 elfcore(vnode_t *vp, proc_t *p, cred_t *credp, rlim64_t rlimit, int sig,
1723     core_content_t content)
1724 {
1725         offset_t poffset, soffset;
1726         Off doffset;
1727         int error, i, nphdrs, nshdrs;
1728         int overflow = 0;
1729         struct seg *seg;
1730         struct as *as = p->p_as;
1731         union {
1732                 Ehdr ehdr;
1733                 Phdr phdr[1];
1734                 Shdr shdr[1];
1735         } *bigwad;
1736         size_t bigsize;
1737         size_t phdrsz, shdrsz;
1738         Ehdr *ehdr;
1739         Phdr *v;
1740         caddr_t brkbase;
1741         size_t brksize;
1742         caddr_t stkbase;
1743         size_t stksize;
1744         int ntries = 0;
1745         klwp_t *lwp = ttolwp(curthread);

1747 top:
1748         /*
1749          * Make sure we have everything we need (registers, etc.).
1750          * All other lwps have already stopped and are in an orderly state.
1751          */
1752         ASSERT(p == ttoproc(curthread));
1753         prstop(0, 0);

1755         AS_LOCK_ENTER(as, RW_WRITER);
1755         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1756         nphdrs = prnsegs(as, 0) + 2;            /* two CORE note sections */

1758         /*
1759          * Count the number of section headers we're going to need.
1760          */
1761         nshdrs = 0;
1762         if (content & (CC_CONTENT_CTF | CC_CONTENT_SYMTAB)) {
1763                 (void) process_scns(content, p, credp, NULL, NULL, NULL, 0,
1764                     NULL, &nshdrs);
1765         }
1766         AS_LOCK_EXIT(as);
1766         AS_LOCK_EXIT(as, &as->a_lock);

1768         ASSERT(nshdrs == 0 || nshdrs > 1);

1770         /*
1771          * The core file contents may required zero section headers, but if
1772          * we overflow the 16 bits allotted to the program header count in
1773          * the ELF header, we'll need that program header at index zero.
1774          */
1775         if (nshdrs == 0 && nphdrs >= PN_XNUM)
1776                 nshdrs = 1;
```



```
1778         phdrsz = nphdrs * sizeof (Phdr);
1779         shdrsz = nshdrs * sizeof (Shdr);

1781         bigsize = MAX(sizeof (*bigwad), MAX(phdrsz, shdrsz));
1782         bigwad = kmem_alloc(bigsize, KM_SLEEP);

1784         ehdr = &bigwad->ehdr;
1785         bzero(ehdr, sizeof (*ehdr));

1787         ehdr->e_ident[EI_MAG0] = ELFMAG0;
1788         ehdr->e_ident[EI_MAG1] = ELFMAG1;
1789         ehdr->e_ident[EI_MAG2] = ELFMAG2;
1790         ehdr->e_ident[EI_MAG3] = ELFMAG3;
1791         ehdr->e_ident[EI_CLASS] = ELFCLASS;
1792         ehdr->e_type = ET_CORE;

1794 #if !defined(_LP64) || defined(_ELF32_COMPAT)

1796 #if defined(__sparc)
1797         ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1798         ehdr->e_machine = EM_SPARC;
1799 #elif defined(__i386) || defined(__i386_COMPAT)
1800         ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1801         ehdr->e_machine = EM_386;
1802 #else
1803 #error "no recognized machine type is defined"
1804 #endif

1806 #else   /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1808 #if defined(__sparc)
1809         ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1810         ehdr->e_machine = EM_SPARCV9;
1811 #elif defined(__amd64)
1812         ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1813         ehdr->e_machine = EM_AMD64;
1814 #else
1815 #error "no recognized 64-bit machine type is defined"
1816 #endif

1818 #endif  /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1820         /*
1821          * If the count of program headers or section headers or the index
1822          * of the section string table can't fit in the mere 16 bits
1823          * shortsightedly allotted to them in the ELF header, we use the
1824          * extended formats and put the real values in the section header
1825          * as index 0.
1826          */
1827         ehdr->e_version = EV_CURRENT;
1828         ehdr->e_ehsize = sizeof (Ehdr);

1830         if (nphdrs >= PN_XNUM)
1831                 ehdr->e_phnum = PN_XNUM;
1832         else
1833                 ehdr->e_phnum = (unsigned short)nphdrs;

1835         ehdr->e_phoff = sizeof (Ehdr);
1836         ehdr->e_phentsize = sizeof (Phdr);

1838         if (nshdrs > 0) {
1839                 if (nshdrs >= SHN_LORESERVE)
1840                         ehdr->e_shnum = 0;
1841                 else
1842                         ehdr->e_shnum = (unsigned short)nshdrs;
```

```
1844                if (nshdrs - 1 >= SHN_LORESERVE)
1845                        ehdr->e_shstrndx = SHN_XINDEX;
1846                else
1847                        ehdr->e_shstrndx = (unsigned short)(nshdrs - 1);

1849                ehdr->e_shoff = ehdr->e_phoff + ehdr->e_phentsize * nphdrs;
1850                ehdr->e_shentsize = sizeof (Shdr);
1851        }

1853        if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, ehdr,
1854            sizeof (Ehdr), rlimit, credp))
1855                goto done;

1857        poffset = sizeof (Ehdr);
1858        soffset = sizeof (Ehdr) + phdrsz;
1859        doffset = sizeof (Ehdr) + phdrsz + shdrsz;

1861        v = &bigwad->phdr[0];
1862        bzero(v, phdrsz);

1864        setup_old_note_header(&v[0], p);
1865        v[0].p_offset = doffset = roundup(doffset, sizeof (Word));
1866        doffset += v[0].p_filesz;

1868        setup_note_header(&v[1], p);
1869        v[1].p_offset = doffset = roundup(doffset, sizeof (Word));
1870        doffset += v[1].p_filesz;

1872        mutex_enter(&p->p_lock);

1874        brkbase = p->p_brkbase;
1875        brksize = p->p_brksize;

1877        stkbase = p->p_usrstack - p->p_stksize;
1878        stksize = p->p_stksize;

1880        mutex_exit(&p->p_lock);

1882        AS_LOCK_ENTER(as, RW_WRITER);
1882        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1883        i = 2;
1884        for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1885                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1886                caddr_t saddr, naddr;
1887                void *tmp = NULL;
1888                extern struct seg_ops segspt_shmops;

1890                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1891                        uint_t prot;
1892                        size_t size;
1893                        int type;
1894                        vnode_t *mvp;

1896                        prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1897                        prot &= PROT_READ | PROT_WRITE | PROT_EXEC;
1898                        if ((size = (size_t)(naddr - saddr)) == 0)
1899                                continue;
1900                        if (i == nphdrs) {
1901                                overflow++;
1902                                continue;
1903                        }
1904                        v[i].p_type = PT_LOAD;
1905                        v[i].p_vaddr = (Addr)(uintptr_t)saddr;
1906                        v[i].p_memsz = size;
1907                        if (prot & PROT_READ)
1908                                v[i].p_flags |= PF_R;
```



```
1909                        if (prot & PROT_WRITE)
1910                                v[i].p_flags |= PF_W;
1911                        if (prot & PROT_EXEC)
1912                                v[i].p_flags |= PF_X;

1914                        /*
1915                         * Figure out which mappings to include in the core.
1916                         */
1917                        type = SEGOP_GETTYPE(seg, saddr);

1919                        if (saddr == stkbase && size == stksize) {
1920                                if (!(content & CC_CONTENT_STACK))
1921                                        goto exclude;

1923                        } else if (saddr == brkbase && size == brksize) {
1924                                if (!(content & CC_CONTENT_HEAP))
1925                                        goto exclude;

1927                        } else if (seg->s_ops == &segspt_shmops) {
1928                                if (type & MAP_NORESERVE) {
1929                                        if (!(content & CC_CONTENT_DISM))
1930                                                goto exclude;
1931                                } else {
1932                                        if (!(content & CC_CONTENT_ISM))
1933                                                goto exclude;
1934                                }

1936                        } else if (seg->s_ops != &segvn_ops) {
1937                                goto exclude;

1939                        } else if (type & MAP_SHARED) {
1940                                if (shmgetid(p, saddr) != SHMID_NONE) {
1941                                        if (!(content & CC_CONTENT_SHM))
1942                                                goto exclude;

1944                                } else if (SEGOP_GETVP(seg, seg->s_base,
1945                                    &mvp) != 0 || mvp == NULL ||
1946                                    mvp->v_type != VREG) {
1947                                        if (!(content & CC_CONTENT_SHANON))
1948                                                goto exclude;

1950                                } else {
1951                                        if (!(content & CC_CONTENT_SHFILE))
1952                                                goto exclude;
1953                                }

1955                        } else if (SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
1956                            mvp == NULL || mvp->v_type != VREG) {
1957                                if (!(content & CC_CONTENT_ANON))
1958                                        goto exclude;

1960                        } else if (prot == (PROT_READ | PROT_EXEC)) {
1961                                if (!(content & CC_CONTENT_TEXT))
1962                                        goto exclude;

1964                        } else if (prot == PROT_READ) {
1965                                if (!(content & CC_CONTENT_RODATA))
1966                                        goto exclude;

1968                        } else {
1969                                if (!(content & CC_CONTENT_DATA))
1970                                        goto exclude;
1971                        }

1973                        doffset = roundup(doffset, sizeof (Word));
1974                        v[i].p_offset = doffset;
```

```
1975                        v[i].p_filesz = size;
1976                        doffset += size;
1977 exclude:
1978                        i++;
1979                }
1980                ASSERT(tmp == NULL);
1981        }
1982        AS_LOCK_EXIT(as);
1982        AS_LOCK_EXIT(as, &as->a_lock);

1984        if (overflow || i != nphdrs) {
1985                if (ntries++ == 0) {
1986                        kmem_free(bigwad, bigsize);
1987                        overflow = 0;
1988                        goto top;
1989                }
1990                cmn_err(CE_WARN, "elfcore: core dump failed for "
1991                    "process %d; address space is changing", p->p_pid);
1992                error = EIO;
1993                goto done;
1994        }

1996        if ((error = core_write(vp, UIO_SYSSPACE, poffset,
1997            v, phdrsz, rlimit, credp)) != 0)
1998                goto done;

2000        if ((error = write_old_elfnotes(p, sig, vp, v[0].p_offset, rlimit,
2001            credp)) != 0)
2002                goto done;

2004        if ((error = write_elfnotes(p, sig, vp, v[1].p_offset, rlimit,
2005            credp, content)) != 0)
2006                goto done;

2008        for (i = 2; i < nphdrs; i++) {
2009                prkillinfo_t killinfo;
2010                sigqueue_t *sq;
2011                int sig, j;

2013                if (v[i].p_filesz == 0)
2014                        continue;

2016                /*
2017                 * If dumping out this segment fails, rather than failing
2018                 * the core dump entirely, we reset the size of the mapping
2019                 * to zero to indicate that the data is absent from the core
2020                 * file and or in the PF_SUNW_FAILURE flag to differentiate
2021                 * this from mappings that were excluded due to the core file
2022                 * content settings.
2023                 */
2024                if ((error = core_seg(p, vp, v[i].p_offset,
2025                    (caddr_t)(uintptr_t)v[i].p_vaddr, v[i].p_filesz,
2026                    rlimit, credp)) == 0) {
2027                        continue;
2028                }

2030                if ((sig = lwp->lwp_cursig) == 0) {
2031                        /*
2032                         * We failed due to something other than a signal.
2033                         * Since the space reserved for the segment is now
2034                         * unused, we stash the errno in the first four
2035                         * bytes. This undocumented interface will let us
2036                         * understand the nature of the failure.
2037                         */
2038                        (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2039                            &error, sizeof (error), rlimit, credp);
```



```
2041                        v[i].p_filesz = 0;
2042                        v[i].p_flags |= PF_SUNW_FAILURE;
2043                        if ((error = core_write(vp, UIO_SYSSPACE,
2044                            poffset + sizeof (v[i]) * i, &v[i], sizeof (v[i]),
2045                            rlimit, credp)) != 0)
2046                                goto done;

2048                        continue;
2049                }

2051                /*
2052                 * We took a signal.  We want to abort the dump entirely, but
2053                 * we also want to indicate what failed and why.  We therefore
2054                 * use the space reserved for the first failing segment to
2055                 * write our error (which, for purposes of compatability with
2056                 * older core dump readers, we set to EINTR) followed by any
2057                 * siginfo associated with the signal.
2058                 */
2059                bzero(&killinfo, sizeof (killinfo));
2060                killinfo.prk_error = EINTR;

2062                sq = sig == SIGKILL ? curproc->p_killsqp : lwp->lwp_curinfo;

2064                if (sq != NULL) {
2065                        bcopy(&sq->sq_info, &killinfo.prk_info,
2066                            sizeof (sq->sq_info));
2067                } else {
2068                        killinfo.prk_info.si_signo = lwp->lwp_cursig;
2069                        killinfo.prk_info.si_code = SI_NOINFO;
2070                }

2072 #if (defined(_SYSCALL32_IMPL) || defined(_LP64))
2073                /*
2074                 * If this is a 32-bit process, we need to translate from the
2075                 * native siginfo to the 32-bit variant.  (Core readers must
2076                 * always have the same data model as their target or must
2077                 * be aware of -- and compensate for -- data model differences.)
2078                 */
2079                if (curproc->p_model == DATAMODEL_ILP32) {
2080                        siginfo32_t si32;

2082                        siginfo_kto32((k_siginfo_t *)&killinfo.prk_info, &si32);
2083                        bcopy(&si32, &killinfo.prk_info, sizeof (si32));
2084                }
2085 #endif

2087                (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2088                    &killinfo, sizeof (killinfo), rlimit, credp);

2090                /*
2091                 * For the segment on which we took the signal, indicate that
2092                 * its data now refers to a siginfo.
2093                 */
2094                v[i].p_filesz = 0;
2095                v[i].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED |
2096                    PF_SUNW_SIGINFO;

2098                /*
2099                 * And for every other segment, indicate that its absence
2100                 * is due to a signal.
2101                 */
2102                for (j = i + 1; j < nphdrs; j++) {
2103                        v[j].p_filesz = 0;
2104                        v[j].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED;
2105                }
```

```
2107                 /*
2108                  * Finally, write out our modified program headers.
2109                  */
2110                 if ((error = core_write(vp, UIO_SYSSPACE,
2111                     poffset + sizeof (v[i]) * i, &v[i],
2112                     sizeof (v[i]) * (nphdrs - i), rlimit, credp)) != 0)
2113                         goto done;

2115                 break;
2116         }

2118         if (nshdrs > 0) {
2119                 bzero(&bigwad->shdr[0], shdrsz);

2121                 if (nshdrs >= SHN_LORESERVE)
2122                         bigwad->shdr[0].sh_size = nshdrs;

2124                 if (nshdrs - 1 >= SHN_LORESERVE)
2125                         bigwad->shdr[0].sh_link = nshdrs - 1;

2127                 if (nphdrs >= PN_XNUM)
2128                         bigwad->shdr[0].sh_info = nphdrs;

2130                 if (nshdrs > 1) {
2131                         AS_LOCK_ENTER(as, RW_WRITER);
2131                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2132                         if ((error = process_scns(content, p, credp, vp,
2133                             &bigwad->shdr[0], nshdrs, rlimit, &doffset,
2134                             NULL)) != 0) {
2135                                 AS_LOCK_EXIT(as);
2135                                 AS_LOCK_EXIT(as, &as->a_lock);
2136                                 goto done;
2137                         }
2138                         AS_LOCK_EXIT(as);
2138                         AS_LOCK_EXIT(as, &as->a_lock);
2139                 }

2141                 if ((error = core_write(vp, UIO_SYSSPACE, soffset,
2142                     &bigwad->shdr[0], shdrsz, rlimit, credp)) != 0)
2143                         goto done;
2144         }

2146 done:
2147         kmem_free(bigwad, bigsize);
2148         return (error);
2149 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   87000 Wed Nov 25 13:59:35 2015
new/usr/src/uts/common/fs/doorfs/door_sys.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

2984 /*
2985  * Copy data from 'src' in current address space to 'dest' in 'as' for 'len'
2986  * bytes.
2987  *
2988  * Performs this using 1 mapin and 1 copy operation.
2989  *
2990  * We really should do more than 1 page at a time to improve
2991  * performance, but for now this is treated as an anomalous condition.
2992  */
2993 static int
2994 door_copy(struct as *as, caddr_t src, caddr_t dest, uint_t len)
2995 {
2996         caddr_t kaddr;
2997         caddr_t rdest;
2998         uint_t  off;
2999         page_t  **pplist;
3000         page_t  *pp = NULL;
3001         int     error = 0;

3003         ASSERT(len <= PAGESIZE);
3004         off = (uintptr_t)dest & PAGEOFFSET;     /* offset within the page */
3005         rdest = (caddr_t)((uintptr_t)dest &
3006             (uintptr_t)PAGEMASK);       /* Page boundary */
3007         ASSERT(off + len <= PAGESIZE);

3009         /*
3010          * Lock down destination page.
3011          */
3012         if (as_pagelock(as, &pplist, rdest, PAGESIZE, S_WRITE))
3013                 return (E2BIG);
3014         /*
3015          * Check if we have a shadow page list from as_pagelock. If not,
3016          * we took the slow path and have to find our page struct the hard
3017          * way.
3018          */
3019         if (pplist == NULL) {
3020                 pfn_t   pfnum;

3022                 /* MMU mapping is already locked down */
3023                 AS_LOCK_ENTER(as, RW_READER);
3023                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3024                 pfnum = hat_getpfnum(as->a_hat, rdest);
3025                 AS_LOCK_EXIT(as);
3025                 AS_LOCK_EXIT(as, &as->a_lock);

3027                 /*
3028                  * TODO: The pfn step should not be necessary - need
3029                  * a hat_getpp() function.
3030                  */
3031                 if (pf_is_memory(pfnum)) {
3032                         pp = page_numtopp_nolock(pfnum);
3033                         ASSERT(pp == NULL || PAGE_LOCKED(pp));
3034                 } else
3035                         pp = NULL;
3036                 if (pp == NULL) {
3037                         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3038                         return (E2BIG);
3039                 }
3040         } else {
```



```
3041                 pp = *pplist;
3042         }
3043         /*
3044          * Map destination page into kernel address
3045          */
3046         if (kpm_enable)
3047                 kaddr = (caddr_t)hat_kpm_mapin(pp, (struct kpme *)NULL);
3048         else
3049                 kaddr = (caddr_t)ppmapin(pp, PROT_READ | PROT_WRITE,
3050                     (caddr_t)-1);

3052         /*
3053          * Copy from src to dest
3054          */
3055         if (copyin_nowatch(src, kaddr + off, len) != 0)
3056                 error = EFAULT;
3057         /*
3058          * Unmap destination page from kernel
3059          */
3060         if (kpm_enable)
3061                 hat_kpm_mapout(pp, (struct kpme *)NULL, kaddr);
3062         else
3063                 ppmapout(kaddr);
3064         /*
3065          * Unlock destination page
3066          */
3067         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3068         return (error);
3069 }
______unchanged_portion_omitted_
```

```
**********************************************************
   58186 Wed Nov 25 13:59:35 2015
new/usr/src/uts/common/fs/proc/prcontrol.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

2442 /*
2443  * Cancel all watched areas.  Called from prclose().
2444  */
2445 proc_t *
2446 pr_cancel_watch(prnode_t *pnp)
2447 {
2448         proc_t *p = pnp->pr_pcommon->prc_proc;
2449         struct as *as;
2450         kthread_t *t;

2452         ASSERT(MUTEX_HELD(&p->p_lock) && (p->p_proc_flag & P_PR_LOCK));

2454         if (!pr_watch_active(p))
2455                 return (p);

2457         /*
2458          * Pause the process before dealing with the watchpoints.
2459          */
2460         if (p == curproc) {
2461                 prunlock(pnp);
2462                 while (holdwatch() != 0)
2463                         continue;
2464                 p = pr_p_lock(pnp);
2465                 mutex_exit(&pr_pidlock);
2466                 ASSERT(p == curproc);
2467         } else {
2468                 pauselwps(p);
2469                 while (p != NULL && pr_allstopped(p, 0) > 0) {
2470                         /*
2471                          * This cv/mutex pair is persistent even
2472                          * if the process disappears after we
2473                          * unmark it and drop p->p_lock.
2474                          */
2475                         kcondvar_t *cv = &pr_pid_cv[p->p_slot];
2476                         kmutex_t *mp = &p->p_lock;

2478                         prunmark(p);
2479                         (void) cv_wait(cv, mp);
2480                         mutex_exit(mp);
2481                         p = pr_p_lock(pnp);  /* NULL if process disappeared */
2482                         mutex_exit(&pr_pidlock);
2483                 }
2484         }

2486         if (p == NULL)          /* the process disappeared */
2487                 return (NULL);

2489         ASSERT(p == pnp->pr_pcommon->prc_proc);
2490         ASSERT(MUTEX_HELD(&p->p_lock) && (p->p_proc_flag & P_PR_LOCK));

2492         if (pr_watch_active(p)) {
2493                 pr_free_watchpoints(p);
2494                 if ((t = p->p_tlist) != NULL) {
2495                         do {
2496                                 watch_disable(t);

2498                         } while ((t = t->t_forw) != p->p_tlist);
2499                 }
2500         }
```



```
2502         if ((as = p->p_as) != NULL) {
2503                 avl_tree_t *tree;
2504                 struct watched_page *pwp;

2506                 /*
2507                  * If this is the parent of a vfork, the watched page
2508                  * list has been moved temporarily to p->p_wpage.
2509                  */
2510                 if (avl_numnodes(&p->p_wpage) != 0)
2511                         tree = &p->p_wpage;
2512                 else
2513                         tree = &as->a_wpage;

2515                 mutex_exit(&p->p_lock);
2516                 AS_LOCK_ENTER(as, RW_WRITER);
2516                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

2518                 for (pwp = avl_first(tree); pwp != NULL;
2519                     pwp = AVL_NEXT(tree, pwp)) {
2520                         pwp->wp_read = 0;
2521                         pwp->wp_write = 0;
2522                         pwp->wp_exec = 0;
2523                         if ((pwp->wp_flags & WP_SETPROT) == 0) {
2524                                 pwp->wp_flags |= WP_SETPROT;
2525                                 pwp->wp_prot = pwp->wp_oprot;
2526                                 pwp->wp_list = p->p_wprot;
2527                                 p->p_wprot = pwp;
2528                         }
2529                 }

2531                 AS_LOCK_EXIT(as);
2531                 AS_LOCK_EXIT(as, &as->a_lock);
2532                 mutex_enter(&p->p_lock);
2533         }

2535         /*
2536          * Unpause the process now.
2537          */
2538         if (p == curproc)
2539                 continuelwps(p);
2540         else
2541                 unpauselwps(p);

2543         return (p);
2544 }
______unchanged_portion_omitted_
```

```
**********************************************************
   93542 Wed Nov 25 13:59:36 2015
new/usr/src/uts/common/fs/proc/prioctl.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_


 132 /*
 133  * Control operations (lots).
 134  */
 135 /*ARGSUSED*/
 136 #ifdef _SYSCALL32_IMPL
 137 static int
 138 prioctl64(
 139         struct vnode *vp,
 140         int cmd,
 141         intptr_t arg,
 142         int flag,
 143         cred_t *cr,
 144         int *rvalp,
 145         caller_context_t *ct)
 146 #else
 147 int
 148 prioctl(
 149         struct vnode *vp,
 150         int cmd,
 151         intptr_t arg,
 152         int flag,
 153         cred_t *cr,
 154         int *rvalp,
 155         caller_context_t *ct)
 156 #endif  /* _SYSCALL32_IMPL */
 157 {
 158         int nsig = PROC_IS_BRANDED(curproc)? BROP(curproc)->b_nsig : NSIG;
 159         caddr_t cmaddr = (caddr_t)arg;
 160         proc_t *p;
 161         user_t *up;
 162         kthread_t *t;
 163         klwp_t *lwp;
 164         prnode_t *pnp = VTOP(vp);
 165         prcommon_t *pcp;
 166         prnode_t *xpnp = NULL;
 167         int error;
 168         int zdisp;
 169         void *thing = NULL;
 170         size_t thingsize = 0;

 172         /*
 173          * For copyin()/copyout().
 174          */
 175         union {
 176                 caddr_t         va;
 177                 int             signo;
 178                 int             nice;
 179                 uint_t          lwpid;
 180                 long            flags;
 181                 prstatus_t      prstat;
 182                 prrun_t         prrun;
 183                 sigset_t        smask;
 184                 siginfo_t       info;
 185                 sysset_t        prmask;
 186                 prgregset_t     regs;
 187                 prfpregset_t    fpregs;
 188                 prpsinfo_t      prps;
 189                 sigset_t        holdmask;
```



```
 190                 fltset_t        fltmask;
 191                 prcred_t        prcred;
 192                 prhusage_t      prhusage;
 193                 prmap_t         prmap;
 194                 auxv_t          auxv[__KERN_NAUXV_IMPL];
 195         } un;

 197         if (pnp->pr_type == PR_TMPL)
 198                 return (prctioctl(pnp, cmd, arg, flag, cr));

 200         /*
 201          * Support for old /proc interface.
 202          */
 203         if (pnp->pr_pidfile != NULL) {
 204                 ASSERT(pnp->pr_type == PR_PIDDIR);
 205                 vp = pnp->pr_pidfile;
 206                 pnp = VTOP(vp);
 207                 ASSERT(pnp->pr_type == PR_PIDFILE);
 208         }

 210         if (pnp->pr_type != PR_PIDFILE && pnp->pr_type != PR_LWPIDFILE)
 211                 return (ENOTTY);

 213         /*
 214          * Fail ioctls which are logically "write" requests unless
 215          * the user has write permission.
 216          */
 217         if ((flag & FWRITE) == 0 && isprwrioctl(cmd))
 218                 return (EBADF);

 220         /*
 221          * Perform any necessary copyin() operations before
 222          * locking the process.  Helps avoid deadlocks and
 223          * improves performance.
 224          *
 225          * Also, detect invalid ioctl codes here to avoid
 226          * locking a process unnnecessarily.
 227          *
 228          * Also, prepare to allocate space that will be needed below,
 229          * case by case.
 230          */
 231         error = 0;
 232         switch (cmd) {
 233         case PIOCGETPR:
 234                 thingsize = sizeof (proc_t);
 235                 break;
 236         case PIOCGETU:
 237                 thingsize = sizeof (user_t);
 238                 break;
 239         case PIOCSTOP:
 240         case PIOCWSTOP:
 241         case PIOCLWPIDS:
 242         case PIOCGTRACE:
 243         case PIOCGENTRY:
 244         case PIOCGEXIT:
 245         case PIOCSRLC:
 246         case PIOCRRLC:
 247         case PIOCSFORK:
 248         case PIOCRFORK:
 249         case PIOCGREG:
 250         case PIOCGFPREG:
 251         case PIOCSTATUS:
 252         case PIOCLSTATUS:
 253         case PIOCPSINFO:
 254         case PIOCMAXSIG:
 255         case PIOCGXREGSIZE:
```

```
256                 break;
257         case PIOCSXREG:         /* set extra registers */
258         case PIOCGXREG:         /* get extra registers */
259 #if defined(__sparc)
260                 thingsize = sizeof (prxregset_t);
261 #else
262                 thingsize = 0;
263 #endif
264                 break;
265         case PIOCACTION:
266                 thingsize = (nsig-1) * sizeof (struct sigaction);
267                 break;
268         case PIOCGHOLD:
269         case PIOCNMAP:
270         case PIOCMAP:
271         case PIOCGFAULT:
272         case PIOCCFAULT:
273         case PIOCCRED:
274         case PIOCGROUPS:
275         case PIOCUSAGE:
276         case PIOCLUSAGE:
277                 break;
278         case PIOCOPENPD:
279                 /*
280                  * We will need this below.
281                  * Allocate it now, before locking the process.
282                  */
283                 xpnp = prgetnode(vp, PR_OPAGEDATA);
284                 break;
285         case PIOCNAUXV:
286         case PIOCAUXV:
287                 break;

289 #if defined(__i386) || defined(__amd64)
290         case PIOCNLDT:
291         case PIOCLDT:
292                 break;
293 #endif  /* __i386 || __amd64 */

295 #if defined(__sparc)
296         case PIOCGWIN:
297                 thingsize = sizeof (gwindows_t);
298                 break;
299 #endif  /* __sparc */

301         case PIOCOPENM:         /* open mapped object for reading */
302                 if (cmaddr == NULL)
303                         un.va = NULL;
304                 else if (copyin(cmaddr, &un.va, sizeof (un.va)))
305                         error = EFAULT;
306                 break;

308         case PIOCRUN:           /* make lwp or process runnable */
309                 if (cmaddr == NULL)
310                         un.prrun.pr_flags = 0;
311                 else if (copyin(cmaddr, &un.prrun, sizeof (un.prrun)))
312                         error = EFAULT;
313                 break;

315         case PIOCOPENLWP:       /* return /proc lwp file descriptor */
316                 if (copyin(cmaddr, &un.lwpid, sizeof (un.lwpid)))
317                         error = EFAULT;
318                 break;

320         case PIOCSTRACE:        /* set signal trace mask */
321                 if (copyin(cmaddr, &un.smask, sizeof (un.smask)))
```



```
322                         error = EFAULT;
323                 break;

325         case PIOCSSIG:          /* set current signal */
326                 if (cmaddr == NULL)
327                         un.info.si_signo = 0;
328                 else if (copyin(cmaddr, &un.info, sizeof (un.info)))
329                         error = EFAULT;
330                 break;

332         case PIOCKILL:          /* send signal */
333         case PIOCUNKILL:        /* delete a signal */
334                 if (copyin(cmaddr, &un.signo, sizeof (un.signo)))
335                         error = EFAULT;
336                 break;

338         case PIOCNICE:          /* set nice priority */
339                 if (copyin(cmaddr, &un.nice, sizeof (un.nice)))
340                         error = EFAULT;
341                 break;

343         case PIOCSENTRY:        /* set syscall entry bit mask */
344         case PIOCSEXIT:         /* set syscall exit bit mask */
345                 if (copyin(cmaddr, &un.prmask, sizeof (un.prmask)))
346                         error = EFAULT;
347                 break;

349         case PIOCSET:           /* set process flags */
350         case PIOCRESET:         /* reset process flags */
351                 if (copyin(cmaddr, &un.flags, sizeof (un.flags)))
352                         error = EFAULT;
353                 break;

355         case PIOCSREG:          /* set general registers */
356                 if (copyin(cmaddr, un.regs, sizeof (un.regs)))
357                         error = EFAULT;
358                 break;

360         case PIOCSFPREG:        /* set floating-point registers */
361                 if (copyin(cmaddr, &un.fpregs, sizeof (un.fpregs)))
362                         error = EFAULT;
363                 break;

365         case PIOCSHOLD:         /* set signal-hold mask */
366                 if (copyin(cmaddr, &un.holdmask, sizeof (un.holdmask)))
367                         error = EFAULT;
368                 break;

370         case PIOCSFAULT:        /* set mask of traced faults */
371                 if (copyin(cmaddr, &un.fltmask, sizeof (un.fltmask)))
372                         error = EFAULT;
373                 break;

375         default:
376                 error = EINVAL;
377                 break;
378         }

380         if (error)
381                 return (error);

383 startover:
384         /*
385          * If we need kmem_alloc()d space then we allocate it now, before
386          * grabbing the process lock.  Using kmem_alloc(KM_SLEEP) while
387          * holding the process lock leads to deadlock with the clock thread.
```

```
388          * (The clock thread wakes up the pageout daemon to free up space.
389          * If the clock thread blocks behind us and we are sleeping waiting
390          * for space, then space may never become available.)
391          */
392         if (thingsize) {
393                 ASSERT(thing == NULL);
394                 thing = kmem_alloc(thingsize, KM_SLEEP);
395         }

397         switch (cmd) {
398         case PIOCPSINFO:
399         case PIOCGETPR:
400         case PIOCUSAGE:
401         case PIOCLUSAGE:
402                 zdisp = ZYES;
403                 break;
404         case PIOCSXREG:         /* set extra registers */
405                 /*
406                  * perform copyin before grabbing the process lock
407                  */
408                 if (thing) {
409                         if (copyin(cmaddr, thing, thingsize)) {
410                                 kmem_free(thing, thingsize);
411                                 return (EFAULT);
412                         }
413                 }
414                 /* fall through... */
415         default:
416                 zdisp = ZNO;
417                 break;
418         }

420         if ((error = prlock(pnp, zdisp)) != 0) {
421                 if (thing != NULL)
422                         kmem_free(thing, thingsize);
423                 if (xpnp)
424                         prfreenode(xpnp);
425                 return (error);
426         }

428         pcp = pnp->pr_common;
429         p = pcp->prc_proc;
430         ASSERT(p != NULL);

432         /*
433          * Choose a thread/lwp for the operation.
434          */
435         if (zdisp == ZNO && cmd != PIOCSTOP && cmd != PIOCWSTOP) {
436                 if (pnp->pr_type == PR_LWPIDFILE && cmd != PIOCLSTATUS) {
437                         t = pcp->prc_thread;
438                         ASSERT(t != NULL);
439                 } else {
440                         t = prchoose(p);        /* returns locked thread */
441                         ASSERT(t != NULL);
442                         thread_unlock(t);
443                 }
444                 lwp = ttolwp(t);
445         }

447         error = 0;
448         switch (cmd) {

450         case PIOCGETPR:         /* read struct proc */
451         {
452                 proc_t *prp = thing;
```



```
454                 *prp = *p;
455                 prunlock(pnp);
456                 if (copyout(prp, cmaddr, sizeof (proc_t)))
457                         error = EFAULT;
458                 kmem_free(prp, sizeof (proc_t));
459                 thing = NULL;
460                 break;
461         }

463         case PIOCGETU:          /* read u-area */
464         {
465                 user_t *userp = thing;

467                 up = PTOU(p);
468                 *userp = *up;
469                 prunlock(pnp);
470                 if (copyout(userp, cmaddr, sizeof (user_t)))
471                         error = EFAULT;
472                 kmem_free(userp, sizeof (user_t));
473                 thing = NULL;
474                 break;
475         }

477         case PIOCOPENM:         /* open mapped object for reading */
478                 error = propenm(pnp, cmaddr, un.va, rvalp, cr);
479                 /* propenm() called prunlock(pnp) */
480                 break;

482         case PIOCSTOP:          /* stop process or lwp from running */
483         case PIOCWSTOP:         /* wait for process or lwp to stop */
484                 /*
485                  * Can't apply to a system process.
486                  */
487                 if ((p->p_flag & SSYS) || p->p_as == &kas) {
488                         prunlock(pnp);
489                         error = EBUSY;
490                         break;
491                 }

493                 if (cmd == PIOCSTOP)
494                         pr_stop(pnp);

496                 /*
497                  * If an lwp is waiting for itself or its process, don't wait.
498                  * The stopped lwp would never see the fact that it is stopped.
499                  */
500                 if ((pnp->pr_type == PR_LWPIDFILE)?
501                     (pcp->prc_thread == curthread) : (p == curproc)) {
502                         if (cmd == PIOCWSTOP)
503                                 error = EBUSY;
504                         prunlock(pnp);
505                         break;
506                 }

508                 if ((error = pr_wait_stop(pnp, (time_t)0)) != 0)
509                         break;  /* pr_wait_stop() unlocked the process */

511                 if (cmaddr == NULL)
512                         prunlock(pnp);
513                 else {
514                         /*
515                          * Return process/lwp status information.
516                          */
517                         t = pr_thread(pnp);     /* returns locked thread */
518                         thread_unlock(t);
519                         oprgetstatus(t, &un.prstat, VTOZONE(vp));
```

```
520                         prunlock(pnp);
521                         if (copyout(&un.prstat, cmaddr, sizeof (un.prstat)))
522                                 error = EFAULT;
523                 }
524                 break;

526         case PIOCRUN:           /* make lwp or process runnable */
527         {
528                 long flags = un.prrun.pr_flags;

530                 /*
531                  * Cannot set an lwp running is it is not stopped.
532                  * Also, no lwp other than the /proc agent lwp can
533                  * be set running so long as the /proc agent lwp exists.
534                  */
535                 if ((!ISTOPPED(t) && !VSTOPPED(t) &&
536                     !(t->t_proc_flag & TP_PRSTOP)) ||
537                     (p->p_agenttp != NULL &&
538                     (t != p->p_agenttp || pnp->pr_type != PR_LWPIDFILE))) {
539                         prunlock(pnp);
540                         error = EBUSY;
541                         break;
542                 }

544                 if (flags & (PRSHOLD|PRSTRACE|PRSFAULT|PRSVADDR))
545                         prsetrun(t, &un.prrun);

547                 error = pr_setrun(pnp, prmaprunflags(flags));

549                 prunlock(pnp);
550                 break;
551         }

553         case PIOCLWPIDS:        /* get array of lwp identifiers */
554         {
555                 int nlwp;
556                 int Nlwp;
557                 id_t *idp;
558                 id_t *Bidp;

560                 Nlwp = nlwp = p->p_lwpcnt;

562                 if (thing && thingsize != (Nlwp+1) * sizeof (id_t)) {
563                         kmem_free(thing, thingsize);
564                         thing = NULL;
565                 }
566                 if (thing == NULL) {
567                         thingsize = (Nlwp+1) * sizeof (id_t);
568                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
569                 }
570                 if (thing == NULL) {
571                         prunlock(pnp);
572                         goto startover;
573                 }

575                 idp = thing;
576                 thing = NULL;
577                 Bidp = idp;
578                 if ((t = p->p_tlist) != NULL) {
579                         do {
580                                 ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
581                                 ASSERT(nlwp > 0);
582                                 --nlwp;
583                                 *idp++ = t->t_tid;
584                         } while ((t = t->t_forw) != p->p_tlist);
585                 }
```



```
586                 *idp = 0;
587                 ASSERT(nlwp == 0);
588                 prunlock(pnp);
589                 if (copyout(Bidp, cmaddr, (Nlwp+1) * sizeof (id_t)))
590                         error = EFAULT;
591                 kmem_free(Bidp, (Nlwp+1) * sizeof (id_t));
592                 break;
593         }

595         case PIOCOPENLWP:       /* return /proc lwp file descriptor */
596         {
597                 vnode_t *xvp;
598                 int n;

600                 prunlock(pnp);
601                 if ((xvp = prlwpnode(pnp, un.lwpid)) == NULL)
602                         error = ENOENT;
603                 else if (error = fassign(&xvp, flag & (FREAD|FWRITE), &n)) {
604                         VN_RELE(xvp);
605                 } else
606                         *rvalp = n;
607                 break;
608         }

610         case PIOCOPENPD:        /* return /proc page data file descriptor */
611         {
612                 vnode_t *xvp = PTOV(xpnp);
613                 vnode_t *dp = pnp->pr_parent;
614                 int n;

616                 if (pnp->pr_type == PR_LWPIDFILE) {
617                         dp = VTOP(dp)->pr_parent;
618                         dp = VTOP(dp)->pr_parent;
619                 }
620                 ASSERT(VTOP(dp)->pr_type == PR_PIDDIR);

622                 VN_HOLD(dp);
623                 pcp = pnp->pr_pcommon;
624                 xpnp->pr_ino = ptoi(pcp->prc_pid);
625                 xpnp->pr_common = pcp;
626                 xpnp->pr_pcommon = pcp;
627                 xpnp->pr_parent = dp;

629                 xpnp->pr_next = p->p_plist;
630                 p->p_plist = xvp;

632                 prunlock(pnp);
633                 if (error = fassign(&xvp, FREAD, &n)) {
634                         VN_RELE(xvp);
635                 } else
636                         *rvalp = n;

638                 xpnp = NULL;
639                 break;
640         }

642         case PIOCGTRACE:        /* get signal trace mask */
643                 prassignset(&un.smask, &p->p_sigmask);
644                 prunlock(pnp);
645                 if (copyout(&un.smask, cmaddr, sizeof (un.smask)))
646                         error = EFAULT;
647                 break;

649         case PIOCSTRACE:        /* set signal trace mask */
650                 prdelset(&un.smask, SIGKILL);
651                 prassignset(&p->p_sigmask, &un.smask);
```

```
652                 if (!sigisempty(&p->p_sigmask))
653                         p->p_proc_flag |= P_PR_TRACE;
654                 else if (prisempty(&p->p_fltmask)) {
655                         up = PTOU(p);
656                         if (up->u_systrap == 0)
657                                 p->p_proc_flag &= ~P_PR_TRACE;
658                 }
659                 prunlock(pnp);
660                 break;

662         case PIOCSSIG:          /* set current signal */
663                 error = pr_setsig(pnp, &un.info);
664                 prunlock(pnp);
665                 if (un.info.si_signo == SIGKILL && error == 0)
666                         pr_wait_die(pnp);
667                 break;

669         case PIOCKILL:          /* send signal */
670         {
671                 int sig = (int)un.signo;

673                 error = pr_kill(pnp, sig, cr);
674                 prunlock(pnp);
675                 if (sig == SIGKILL && error == 0)
676                         pr_wait_die(pnp);
677                 break;
678         }

680         case PIOCUNKILL:        /* delete a signal */
681                 error = pr_unkill(pnp, (int)un.signo);
682                 prunlock(pnp);
683                 break;

685         case PIOCNICE:          /* set nice priority */
686                 error = pr_nice(p, (int)un.nice, cr);
687                 prunlock(pnp);
688                 break;

690         case PIOCGENTRY:        /* get syscall entry bit mask */
691         case PIOCGEXIT:         /* get syscall exit bit mask */
692                 up = PTOU(p);
693                 if (cmd == PIOCGENTRY) {
694                         prassignset(&un.prmask, &up->u_entrymask);
695                 } else {
696                         prassignset(&un.prmask, &up->u_exitmask);
697                 }
698                 prunlock(pnp);
699                 if (copyout(&un.prmask, cmaddr, sizeof (un.prmask)))
700                         error = EFAULT;
701                 break;

703         case PIOCSENTRY:        /* set syscall entry bit mask */
704         case PIOCSEXIT:         /* set syscall exit bit mask */
705                 pr_setentryexit(p, &un.prmask, cmd == PIOCSENTRY);
706                 prunlock(pnp);
707                 break;

709         case PIOCSRLC:          /* obsolete: set running on last /proc close */
710                 error = pr_set(p, prmapsetflags(PR_RLC));
711                 prunlock(pnp);
712                 break;

714         case PIOCRRLC:          /* obsolete: reset run-on-last-close flag */
715                 error = pr_unset(p, prmapsetflags(PR_RLC));
716                 prunlock(pnp);
717                 break;
```



```
719         case PIOCSFORK:         /* obsolete: set inherit-on-fork flag */
720                 error = pr_set(p, prmapsetflags(PR_FORK));
721                 prunlock(pnp);
722                 break;

724         case PIOCRFORK:         /* obsolete: reset inherit-on-fork flag */
725                 error = pr_unset(p, prmapsetflags(PR_FORK));
726                 prunlock(pnp);
727                 break;

729         case PIOCSET:           /* set process flags */
730                 error = pr_set(p, prmapsetflags(un.flags));
731                 prunlock(pnp);
732                 break;

734         case PIOCRESET:         /* reset process flags */
735                 error = pr_unset(p, prmapsetflags(un.flags));
736                 prunlock(pnp);
737                 break;

739         case PIOCGREG:          /* get general registers */
740                 if (t->t_state != TS_STOPPED && !VSTOPPED(t))
741                         bzero(un.regs, sizeof (un.regs));
742                 else {
743                         /* drop p_lock while touching the lwp's stack */
744                         mutex_exit(&p->p_lock);
745                         prgetprregs(lwp, un.regs);
746                         mutex_enter(&p->p_lock);
747                 }
748                 prunlock(pnp);
749                 if (copyout(un.regs, cmaddr, sizeof (un.regs)))
750                         error = EFAULT;
751                 break;

753         case PIOCSREG:          /* set general registers */
754                 if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
755                         error = EBUSY;
756                 else {
757                         /* drop p_lock while touching the lwp's stack */
758                         mutex_exit(&p->p_lock);
759                         prsetprregs(lwp, un.regs, 0);
760                         mutex_enter(&p->p_lock);
761                 }
762                 prunlock(pnp);
763                 break;

765         case PIOCGFPREG:        /* get floating-point registers */
766                 if (!prhasfp()) {
767                         prunlock(pnp);
768                         error = EINVAL; /* No FP support */
769                         break;
770                 }

772                 if (t->t_state != TS_STOPPED && !VSTOPPED(t))
773                         bzero(&un.fpregs, sizeof (un.fpregs));
774                 else {
775                         /* drop p_lock while touching the lwp's stack */
776                         mutex_exit(&p->p_lock);
777                         prgetprfpregs(lwp, &un.fpregs);
778                         mutex_enter(&p->p_lock);
779                 }
780                 prunlock(pnp);
781                 if (copyout(&un.fpregs, cmaddr, sizeof (un.fpregs)))
782                         error = EFAULT;
783                 break;
```

```
785         case PIOCSFPREG:        /* set floating-point registers */
786                 if (!prhasfp())
787                         error = EINVAL; /* No FP support */
788                 else if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
789                         error = EBUSY;
790                 else {
791                         /* drop p_lock while touching the lwp's stack */
792                         mutex_exit(&p->p_lock);
793                         prsetprfpregs(lwp, &un.fpregs);
794                         mutex_enter(&p->p_lock);
795                 }
796                 prunlock(pnp);
797                 break;

799         case PIOCGXREGSIZE:     /* get the size of the extra registers */
800         {
801                 int xregsize;

803                 if (prhasx(p)) {
804                         xregsize = prgetprxregsize(p);
805                         prunlock(pnp);
806                         if (copyout(&xregsize, cmaddr, sizeof (xregsize)))
807                                 error = EFAULT;
808                 } else {
809                         prunlock(pnp);
810                         error = EINVAL; /* No extra register support */
811                 }
812                 break;
813         }

815         case PIOCGXREG:         /* get extra registers */
816                 if (prhasx(p)) {
817                         bzero(thing, thingsize);
818                         if (t->t_state == TS_STOPPED || VSTOPPED(t)) {
819                                 /* drop p_lock to touch the stack */
820                                 mutex_exit(&p->p_lock);
821                                 prgetprxregs(lwp, thing);
822                                 mutex_enter(&p->p_lock);
823                         }
824                         prunlock(pnp);
825                         if (copyout(thing, cmaddr, thingsize))
826                                 error = EFAULT;
827                 } else {
828                         prunlock(pnp);
829                         error = EINVAL; /* No extra register support */
830                 }
831                 if (thing) {
832                         kmem_free(thing, thingsize);
833                         thing = NULL;
834                 }
835                 break;

837         case PIOCSXREG:         /* set extra registers */
838                 if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
839                         error = EBUSY;
840                 else if (!prhasx(p))
841                         error = EINVAL; /* No extra register support */
842                 else if (thing) {
843                         /* drop p_lock while touching the lwp's stack */
844                         mutex_exit(&p->p_lock);
845                         prsetprxregs(lwp, thing);
846                         mutex_enter(&p->p_lock);
847                 }
848                 prunlock(pnp);
849                 if (thing) {
```



```
850                         kmem_free(thing, thingsize);
851                         thing = NULL;
852                 }
853                 break;

855         case PIOCSTATUS:        /* get process/lwp status */
856                 oprgetstatus(t, &un.prstat, VTOZONE(vp));
857                 prunlock(pnp);
858                 if (copyout(&un.prstat, cmaddr, sizeof (un.prstat)))
859                         error = EFAULT;
860                 break;

862         case PIOCLSTATUS:       /* get status for process & all lwps */
863         {
864                 int Nlwp;
865                 int nlwp;
866                 prstatus_t *Bprsp;
867                 prstatus_t *prsp;

869                 nlwp = Nlwp = p->p_lwpcnt;

871                 if (thing && thingsize != (Nlwp+1) * sizeof (prstatus_t)) {
872                         kmem_free(thing, thingsize);
873                         thing = NULL;
874                 }
875                 if (thing == NULL) {
876                         thingsize = (Nlwp+1) * sizeof (prstatus_t);
877                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
878                 }
879                 if (thing == NULL) {
880                         prunlock(pnp);
881                         goto startover;
882                 }

884                 Bprsp = thing;
885                 thing = NULL;
886                 prsp = Bprsp;
887                 oprgetstatus(t, prsp, VTOZONE(vp));
888                 t = p->p_tlist;
889                 do {
890                         ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
891                         ASSERT(nlwp > 0);
892                         --nlwp;
893                         oprgetstatus(t, ++prsp, VTOZONE(vp));
894                 } while ((t = t->t_forw) != p->p_tlist);
895                 ASSERT(nlwp == 0);
896                 prunlock(pnp);
897                 if (copyout(Bprsp, cmaddr, (Nlwp+1) * sizeof (prstatus_t)))
898                         error = EFAULT;

900                 kmem_free(Bprsp, (Nlwp+1) * sizeof (prstatus_t));
901                 break;
902         }

904         case PIOCPSINFO:        /* get ps(1) information */
905         {
906                 prpsinfo_t *psp = &un.prps;

908                 oprgetpsinfo(p, psp,
909                     (pnp->pr_type == PR_LWPIDFILE)? pcp->prc_thread : NULL);

911                 prunlock(pnp);
912                 if (copyout(&un.prps, cmaddr, sizeof (un.prps)))
913                         error = EFAULT;
914                 break;
915         }
```

```
917         case PIOCMAXSIG:        /* get maximum signal number */
918         {
919                 int n = nsig-1;
921                 prunlock(pnp);
922                 if (copyout(&n, cmaddr, sizeof (n)))
923                         error = EFAULT;
924                 break;
925         }
927         case PIOCACTION:        /* get signal action structures */
928         {
929                 uint_t sig;
930                 struct sigaction *sap = thing;
932                 up = PTOU(p);
933                 for (sig = 1; sig < nsig; sig++)
934                         prgetaction(p, up, sig, &sap[sig-1]);
935                 prunlock(pnp);
936                 if (copyout(sap, cmaddr, (nsig-1) * sizeof (struct sigaction)))
937                         error = EFAULT;
938                 kmem_free(sap, (nsig-1) * sizeof (struct sigaction));
939                 thing = NULL;
940                 break;
941         }
943         case PIOCGHOLD:         /* get signal-hold mask */
944                 schedctl_finish_sigblock(t);
945                 sigktou(&t->t_hold, &un.holdmask);
946                 prunlock(pnp);
947                 if (copyout(&un.holdmask, cmaddr, sizeof (un.holdmask)))
948                         error = EFAULT;
949                 break;
951         case PIOCSHOLD:         /* set signal-hold mask */
952                 pr_sethold(pnp, &un.holdmask);
953                 prunlock(pnp);
954                 break;
956         case PIOCNMAP:          /* get number of memory mappings */
957         {
958                 int n;
959                 struct as *as = p->p_as;
961                 if ((p->p_flag & SSYS) || as == &kas)
962                         n = 0;
963                 else {
964                         mutex_exit(&p->p_lock);
965                         AS_LOCK_ENTER(as, RW_WRITER);
965                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
966                         n = prnsegs(as, 0);
967                         AS_LOCK_EXIT(as);
967                         AS_LOCK_EXIT(as, &as->a_lock);
968                         mutex_enter(&p->p_lock);
969                 }
970                 prunlock(pnp);
971                 if (copyout(&n, cmaddr, sizeof (int)))
972                         error = EFAULT;
973                 break;
974         }
976         case PIOCMAP:           /* get memory map information */
977         {
978                 list_t iolhead;
979                 struct as *as = p->p_as;
```



```
981                 if ((p->p_flag & SSYS) || as == &kas) {
982                         error = 0;
983                         prunlock(pnp);
984                 } else {
985                         mutex_exit(&p->p_lock);
986                         AS_LOCK_ENTER(as, RW_WRITER);
986                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
987                         error = oprgetmap(p, &iolhead);
988                         AS_LOCK_EXIT(as);
988                         AS_LOCK_EXIT(as, &as->a_lock);
989                         mutex_enter(&p->p_lock);
990                         prunlock(pnp);
992                         error = pr_iol_copyout_and_free(&iolhead,
993                             &cmaddr, error);
994                 }
995                 /*
996                  * The procfs PIOCMAP ioctl returns an all-zero buffer
997                  * to indicate the end of the prmap[] array.
998                  * Append it to whatever has already been copied out.
999                  */
1000                bzero(&un.prmap, sizeof (un.prmap));
1001                if (!error && copyout(&un.prmap, cmaddr, sizeof (un.prmap)))
1002                        error = EFAULT;
1004                break;
1005        }
1007        case PIOCGFAULT:        /* get mask of traced faults */
1008                prassignset(&un.fltmask, &p->p_fltmask);
1009                prunlock(pnp);
1010                if (copyout(&un.fltmask, cmaddr, sizeof (un.fltmask)))
1011                        error = EFAULT;
1012                break;
1014        case PIOCSFAULT:        /* set mask of traced faults */
1015                pr_setfault(p, &un.fltmask);
1016                prunlock(pnp);
1017                break;
1019        case PIOCCFAULT:        /* clear current fault */
1020                lwp->lwp_curflt = 0;
1021                prunlock(pnp);
1022                break;
1024        case PIOCCRED:          /* get process credentials */
1025        {
1026                cred_t *cp;
1028                mutex_enter(&p->p_crlock);
1029                cp = p->p_cred;
1030                un.prcred.pr_euid = crgetuid(cp);
1031                un.prcred.pr_ruid = crgetruid(cp);
1032                un.prcred.pr_suid = crgetsuid(cp);
1033                un.prcred.pr_egid = crgetgid(cp);
1034                un.prcred.pr_rgid = crgetrgid(cp);
1035                un.prcred.pr_sgid = crgetsgid(cp);
1036                un.prcred.pr_ngroups = crgetngroups(cp);
1037                mutex_exit(&p->p_crlock);
1039                prunlock(pnp);
1040                if (copyout(&un.prcred, cmaddr, sizeof (un.prcred)))
1041                        error = EFAULT;
1042                break;
1043        }
```

```
1045        case PIOCGROUPS:        /* get supplementary groups */
1046        {
1047                cred_t *cp;

1049                mutex_enter(&p->p_crlock);
1050                cp = p->p_cred;
1051                crhold(cp);
1052                mutex_exit(&p->p_crlock);

1054                prunlock(pnp);
1055                if (copyout(crgetgroups(cp), cmaddr,
1056                    MAX(crgetngroups(cp), 1) * sizeof (gid_t)))
1057                        error = EFAULT;
1058                crfree(cp);
1059                break;
1060        }

1062        case PIOCUSAGE:         /* get usage info */
1063        {
1064                /*
1065                 * For an lwp file descriptor, return just the lwp usage.
1066                 * For a process file descriptor, return total usage,
1067                 * all current lwps plus all defunct lwps.
1068                 */
1069                prhusage_t *pup = &un.prhusage;
1070                prusage_t *upup;

1072                bzero(pup, sizeof (*pup));
1073                pup->pr_tstamp = gethrtime();

1075                if (pnp->pr_type == PR_LWPIDFILE) {
1076                        t = pcp->prc_thread;
1077                        if (t != NULL)
1078                                prgetusage(t, pup);
1079                        else
1080                                error = ENOENT;
1081                } else {
1082                        pup->pr_count  = p->p_defunct;
1083                        pup->pr_create = p->p_mstart;
1084                        pup->pr_term   = p->p_mterm;

1086                        pup->pr_rtime    = p->p_mlreal;
1087                        pup->pr_utime    = p->p_acct[LMS_USER];
1088                        pup->pr_stime    = p->p_acct[LMS_SYSTEM];
1089                        pup->pr_ttime    = p->p_acct[LMS_TRAP];
1090                        pup->pr_tftime   = p->p_acct[LMS_TFAULT];
1091                        pup->pr_dftime   = p->p_acct[LMS_DFAULT];
1092                        pup->pr_kftime   = p->p_acct[LMS_KFAULT];
1093                        pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
1094                        pup->pr_slptime  = p->p_acct[LMS_SLEEP];
1095                        pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
1096                        pup->pr_stoptime = p->p_acct[LMS_STOPPED];

1098                        pup->pr_minf  = p->p_ru.minflt;
1099                        pup->pr_majf  = p->p_ru.majflt;
1100                        pup->pr_nswap = p->p_ru.nswap;
1101                        pup->pr_inblk = p->p_ru.inblock;
1102                        pup->pr_oublk = p->p_ru.oublock;
1103                        pup->pr_msnd  = p->p_ru.msgsnd;
1104                        pup->pr_mrcv  = p->p_ru.msgrcv;
1105                        pup->pr_sigs  = p->p_ru.nsignals;
1106                        pup->pr_vctx  = p->p_ru.nvcsw;
1107                        pup->pr_ictx  = p->p_ru.nivcsw;
1108                        pup->pr_sysc  = p->p_ru.sysc;
1109                        pup->pr_ioch  = p->p_ru.ioch;
```



```
1111                        /*
1112                         * Add the usage information for each active lwp.
1113                         */
1114                        if ((t = p->p_tlist) != NULL &&
1115                            !(pcp->prc_flags & PRC_DESTROY)) {
1116                                do {
1117                                        ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
1118                                        pup->pr_count++;
1119                                        praddusage(t, pup);
1120                                } while ((t = t->t_forw) != p->p_tlist);
1121                        }
1122                }

1124                prunlock(pnp);

1126                upup = kmem_zalloc(sizeof (*upup), KM_SLEEP);
1127                prcvtusage(&un.prhusage, upup);
1128                if (copyout(upup, cmaddr, sizeof (*upup)))
1129                        error = EFAULT;
1130                kmem_free(upup, sizeof (*upup));

1132                break;
1133        }

1135        case PIOCLUSAGE:        /* get detailed usage info */
1136        {
1137                int Nlwp;
1138                int nlwp;
1139                prusage_t *upup;
1140                prusage_t *Bupup;
1141                prhusage_t *pup;
1142                hrtime_t curtime;

1144                nlwp = Nlwp = (pcp->prc_flags & PRC_DESTROY)? 0 : p->p_lwpcnt;

1146                if (thing && thingsize !=
1147                    sizeof (prhusage_t) + (Nlwp+1) * sizeof (prusage_t)) {
1148                        kmem_free(thing, thingsize);
1149                        thing = NULL;
1150                }
1151                if (thing == NULL) {
1152                        thingsize = sizeof (prhusage_t) +
1153                            (Nlwp+1) * sizeof (prusage_t);
1154                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
1155                }
1156                if (thing == NULL) {
1157                        prunlock(pnp);
1158                        goto startover;
1159                }

1161                pup = thing;
1162                upup = Bupup = (prusage_t *)(pup + 1);

1164                ASSERT(p == pcp->prc_proc);

1166                curtime = gethrtime();

1168                /*
1169                 * First the summation over defunct lwps.
1170                 */
1171                bzero(pup, sizeof (*pup));
1172                pup->pr_count  = p->p_defunct;
1173                pup->pr_tstamp = curtime;
1174                pup->pr_create = p->p_mstart;
1175                pup->pr_term   = p->p_mterm;
```

```
1177                pup->pr_rtime    = p->p_mlreal;
1178                pup->pr_utime    = p->p_acct[LMS_USER];
1179                pup->pr_stime    = p->p_acct[LMS_SYSTEM];
1180                pup->pr_ttime    = p->p_acct[LMS_TRAP];
1181                pup->pr_tftime   = p->p_acct[LMS_TFAULT];
1182                pup->pr_dftime   = p->p_acct[LMS_DFAULT];
1183                pup->pr_kftime   = p->p_acct[LMS_KFAULT];
1184                pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
1185                pup->pr_slptime  = p->p_acct[LMS_SLEEP];
1186                pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
1187                pup->pr_stoptime = p->p_acct[LMS_STOPPED];

1189                pup->pr_minf  = p->p_ru.minflt;
1190                pup->pr_majf  = p->p_ru.majflt;
1191                pup->pr_nswap = p->p_ru.nswap;
1192                pup->pr_inblk = p->p_ru.inblock;
1193                pup->pr_oublk = p->p_ru.oublock;
1194                pup->pr_msnd  = p->p_ru.msgsnd;
1195                pup->pr_mrcv  = p->p_ru.msgrcv;
1196                pup->pr_sigs  = p->p_ru.nsignals;
1197                pup->pr_vctx  = p->p_ru.nvcsw;
1198                pup->pr_ictx  = p->p_ru.nivcsw;
1199                pup->pr_sysc  = p->p_ru.sysc;
1200                pup->pr_ioch  = p->p_ru.ioch;

1202                prcvtusage(pup, upup);

1204                /*
1205                 * Fill one prusage struct for each active lwp.
1206                 */
1207                if ((t = p->p_tlist) != NULL &&
1208                    !(pcp->prc_flags & PRC_DESTROY)) {
1209                        do {
1210                                ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
1211                                ASSERT(nlwp > 0);
1212                                --nlwp;
1213                                upup++;
1214                                prgetusage(t, pup);
1215                                prcvtusage(pup, upup);
1216                        } while ((t = t->t_forw) != p->p_tlist);
1217                }
1218                ASSERT(nlwp == 0);

1220                prunlock(pnp);
1221                if (copyout(Bupup, cmaddr, (Nlwp+1) * sizeof (prusage_t)))
1222                        error = EFAULT;
1223                kmem_free(thing, thingsize);
1224                thing = NULL;
1225                break;
1226        }

1228        case PIOCNAUXV:         /* get number of aux vector entries */
1229        {
1230                int n = __KERN_NAUXV_IMPL;

1232                prunlock(pnp);
1233                if (copyout(&n, cmaddr, sizeof (int)))
1234                        error = EFAULT;
1235                break;
1236        }

1238        case PIOCAUXV:          /* get aux vector (see sys/auxv.h) */
1239        {
1240                up = PTOU(p);
1241                bcopy(up->u_auxv, un.auxv,
```



```
1242                    __KERN_NAUXV_IMPL * sizeof (auxv_t));
1243                prunlock(pnp);
1244                if (copyout(un.auxv, cmaddr,
1245                    __KERN_NAUXV_IMPL * sizeof (auxv_t)))
1246                        error = EFAULT;
1247                break;
1248        }

1250 #if defined(__i386) || defined(__amd64)
1251        case PIOCNLDT:          /* get number of LDT entries */
1252        {
1253                int n;

1255                mutex_exit(&p->p_lock);
1256                mutex_enter(&p->p_ldtlock);
1257                n = prnldt(p);
1258                mutex_exit(&p->p_ldtlock);
1259                mutex_enter(&p->p_lock);
1260                prunlock(pnp);
1261                if (copyout(&n, cmaddr, sizeof (n)))
1262                        error = EFAULT;
1263                break;
1264        }

1266        case PIOCLDT:           /* get LDT entries */
1267        {
1268                struct ssd *ssd;
1269                int n;

1271                mutex_exit(&p->p_lock);
1272                mutex_enter(&p->p_ldtlock);
1273                n = prnldt(p);

1275                if (thing && thingsize != (n+1) * sizeof (*ssd)) {
1276                        kmem_free(thing, thingsize);
1277                        thing = NULL;
1278                }
1279                if (thing == NULL) {
1280                        thingsize = (n+1) * sizeof (*ssd);
1281                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
1282                }
1283                if (thing == NULL) {
1284                        mutex_exit(&p->p_ldtlock);
1285                        mutex_enter(&p->p_lock);
1286                        prunlock(pnp);
1287                        goto startover;
1288                }

1290                ssd = thing;
1291                thing = NULL;
1292                if (n != 0)
1293                        prgetldt(p, ssd);
1294                mutex_exit(&p->p_ldtlock);
1295                mutex_enter(&p->p_lock);
1296                prunlock(pnp);

1298                /* mark the end of the list with a null entry */
1299                bzero(&ssd[n], sizeof (*ssd));
1300                if (copyout(ssd, cmaddr, (n+1) * sizeof (*ssd)))
1301                        error = EFAULT;
1302                kmem_free(ssd, (n+1) * sizeof (*ssd));
1303                break;
1304        }
1305 #endif  /* __i386 || __amd64 */

1307 #if defined(__sparc)
```

```
1308         case PIOCGWIN:          /* get gwindows_t (see sys/reg.h) */
1309         {
1310                 gwindows_t *gwp = thing;

1312                 /* drop p->p_lock while touching the stack */
1313                 mutex_exit(&p->p_lock);
1314                 bzero(gwp, sizeof (*gwp));
1315                 prgetwindows(lwp, gwp);
1316                 mutex_enter(&p->p_lock);
1317                 prunlock(pnp);
1318                 if (copyout(gwp, cmaddr, sizeof (*gwp)))
1319                         error = EFAULT;
1320                 kmem_free(gwp, sizeof (gwindows_t));
1321                 thing = NULL;
1322                 break;
1323         }
1324 #endif  /* __sparc */

1326         default:
1327                 prunlock(pnp);
1328                 error = EINVAL;
1329                 break;

1331         }

1333         ASSERT(thing == NULL);
1334         ASSERT(xpnp == NULL);
1335         return (error);
1336 }
```

_____***unchanged_portion_omitted_***

```
1493 void
1494 oprgetpsinfo32(proc_t *p, prpsinfo32_t *psp, kthread_t *tp)
1495 {
1496         kthread_t *t;
1497         char c, state;
1498         user_t *up;
1499         dev_t d;
1500         uint64_t pct;
1501         int retval, niceval;
1502         cred_t *cred;
1503         struct as *as;
1504         hrtime_t hrutime, hrstime, cur_time;

1506         ASSERT(MUTEX_HELD(&p->p_lock));

1508         bzero(psp, sizeof (*psp));

1510         if ((t = tp) == NULL)
1511                 t = prchoose(p);        /* returns locked thread */
1512         else
1513                 thread_lock(t);

1515         /* kludge: map thread state enum into process state enum */

1517         if (t == NULL) {
1518                 state = TS_ZOMB;
1519         } else {
1520                 state = VSTOPPED(t) ? TS_STOPPED : t->t_state;
1521                 thread_unlock(t);
1522         }

1524         switch (state) {
1525         case TS_SLEEP:          state = SSLEEP;         break;
1526         case TS_RUN:            state = SRUN;           break;
1527         case TS_ONPROC:         state = SONPROC;        break;
```



```
1528         case TS_ZOMB:           state = SZOMB;          break;
1529         case TS_STOPPED:        state = SSTOP;          break;
1530         default:                state = 0;              break;
1531         }
1532         switch (state) {
1533         case SSLEEP:    c = 'S';        break;
1534         case SRUN:      c = 'R';        break;
1535         case SZOMB:     c = 'Z';        break;
1536         case SSTOP:     c = 'T';        break;
1537         case SIDL:      c = 'I';        break;
1538         case SONPROC:   c = 'O';        break;
1539 #ifdef SXBRK
1540         case SXBRK:     c = 'X';        break;
1541 #endif
1542         default:        c = '?';        break;
1543         }
1544         psp->pr_state = state;
1545         psp->pr_sname = c;
1546         psp->pr_zomb = (state == SZOMB);
1547         /*
1548          * only export SSYS and SMSACCT; everything else is off-limits to
1549          * userland apps.
1550          */
1551         psp->pr_flag = p->p_flag & (SSYS | SMSACCT);

1553         mutex_enter(&p->p_crlock);
1554         cred = p->p_cred;
1555         psp->pr_uid = crgetruid(cred);
1556         psp->pr_gid = crgetrgid(cred);
1557         psp->pr_euid = crgetuid(cred);
1558         psp->pr_egid = crgetgid(cred);
1559         mutex_exit(&p->p_crlock);

1561         psp->pr_pid = p->p_pid;
1562         if (curproc->p_zone->zone_id != GLOBAL_ZONEID &&
1563             (p->p_flag & SZONETOP)) {
1564                 ASSERT(p->p_zone->zone_id != GLOBAL_ZONEID);
1565                 /*
1566                  * Inside local zones, fake zsched's pid as parent pids for
1567                  * processes which reference processes outside of the zone.
1568                  */
1569                 psp->pr_ppid = curproc->p_zone->zone_zsched->p_pid;
1570         } else {
1571                 psp->pr_ppid = p->p_ppid;
1572         }
1573         psp->pr_pgrp = p->p_pgrp;
1574         psp->pr_sid = p->p_sessp->s_sid;
1575         psp->pr_addr = 0;       /* cannot represent 64-bit addr in 32 bits */
1576         hrutime = mstate_aggr_state(p, LMS_USER);
1577         hrstime = mstate_aggr_state(p, LMS_SYSTEM);
1578         hrt2ts32(hrutime + hrstime, &psp->pr_time);
1579         TICK_TO_TIMESTRUC32(p->p_cutime + p->p_cstime, &psp->pr_ctime);
1580         switch (p->p_model) {
1581         case DATAMODEL_ILP32:
1582                 psp->pr_dmodel = PR_MODEL_ILP32;
1583                 break;
1584         case DATAMODEL_LP64:
1585                 psp->pr_dmodel = PR_MODEL_LP64;
1586                 break;
1587         }
1588         if (state == SZOMB || t == NULL) {
1589                 int wcode = p->p_wcode;         /* must be atomic read */

1591                 if (wcode)
1592                         psp->pr_wstat = wstat(wcode, p->p_wdata);
1593                 psp->pr_lttydev = PRNODEV32;
```

```
1594                psp->pr_ottydev = (o_dev_t)PRNODEV32;
1595                psp->pr_size = 0;
1596                psp->pr_rssize = 0;
1597                psp->pr_pctmem = 0;
1598        } else {
1599                up = PTOU(p);
1600                psp->pr_wchan = 0;      /* cannot represent in 32 bits */
1601                psp->pr_pri = t->t_pri;
1602                (void) strncpy(psp->pr_clname, sclass[t->t_cid].cl_name,
1603                    sizeof (psp->pr_clname) - 1);
1604                retval = CL_DONICE(t, NULL, 0, &niceval);
1605                if (retval == 0) {
1606                        psp->pr_oldpri = v.v_maxsyspri - psp->pr_pri;
1607                        psp->pr_nice = niceval + NZERO;
1608                } else {
1609                        psp->pr_oldpri = 0;
1610                        psp->pr_nice = 0;
1611                }
1612                d = cttydev(p);
1613 #ifdef sun
1614                {
1615                        extern dev_t rwsconsdev, rconsdev, uconsdev;
1616                        /*
1617                         * If the controlling terminal is the real
1618                         * or workstation console device, map to what the
1619                         * user thinks is the console device. Handle case when
1620                         * rwsconsdev or rconsdev is set to NODEV for Starfire.
1621                         */
1622                        if ((d == rwsconsdev || d == rconsdev) && d != NODEV)
1623                                d = uconsdev;
1624                }
1625 #endif
1626                (void) cmpldev(&psp->pr_lttydev, d);
1627                psp->pr_ottydev = cmpdev(d);
1628                TIMESPEC_TO_TIMESPEC32(&psp->pr_start, &up->u_start);
1629                bcopy(up->u_comm, psp->pr_fname,
1630                    MIN(sizeof (up->u_comm), sizeof (psp->pr_fname)-1));
1631                bcopy(up->u_psargs, psp->pr_psargs,
1632                    MIN(PRARGSZ-1, PSARGSZ));
1633                psp->pr_syscall = t->t_sysnum;
1634                psp->pr_argc = up->u_argc;
1635                psp->pr_argv = (caddr32_t)up->u_argv;
1636                psp->pr_envp = (caddr32_t)up->u_envp;

1638                /* compute %cpu for the lwp or process */
1639                pct = 0;
1640                if ((t = tp) == NULL)
1641                        t = p->p_tlist;
1642                cur_time = gethrtime_unscaled();
1643                do {
1644                        pct += cpu_update_pct(t, cur_time);
1645                        if (tp != NULL)         /* just do the one lwp */
1646                                break;
1647                } while ((t = t->t_forw) != p->p_tlist);

1649                psp->pr_pctcpu = prgetpctcpu(pct);
1650                psp->pr_cpu = (psp->pr_pctcpu*100 + 0x6000) >> 15; /* [0..99] */
1651                if (psp->pr_cpu > 99)
1652                        psp->pr_cpu = 99;

1654                if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
1655                        psp->pr_size = 0;
1656                        psp->pr_rssize = 0;
1657                        psp->pr_pctmem = 0;
1658                } else {
1659                        mutex_exit(&p->p_lock);
```



```
1660                        AS_LOCK_ENTER(as, RW_READER);
1660                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1661                        psp->pr_size = (size32_t)btopr(as->a_resvsize);
1662                        psp->pr_rssize = (size32_t)rm_asrss(as);
1663                        psp->pr_pctmem = rm_pctmemory(as);
1664                        AS_LOCK_EXIT(as);
1664                        AS_LOCK_EXIT(as, &as->a_lock);
1665                        mutex_enter(&p->p_lock);
1666                }
1667        }
1668        psp->pr_bysize = (size32_t)ptob(psp->pr_size);
1669        psp->pr_byrssize = (size32_t)ptob(psp->pr_rssize);

1671        /*
1672         * If we are looking at an LP64 process, zero out
1673         * the fields that cannot be represented in ILP32.
1674         */
1675        if (p->p_model != DATAMODEL_ILP32) {
1676                psp->pr_size = 0;
1677                psp->pr_rssize = 0;
1678                psp->pr_bysize = 0;
1679                psp->pr_byrssize = 0;
1680                psp->pr_argv = 0;
1681                psp->pr_envp = 0;
1682        }
1683 }

1685 /*ARGSUSED*/
1686 static int
1687 prioctl32(
1688        struct vnode *vp,
1689        int cmd,
1690        intptr_t arg,
1691        int flag,
1692        cred_t *cr,
1693        int *rvalp,
1694        caller_context_t *ct)
1695 {
1696        int nsig = PROC_IS_BRANDED(curproc)? BROP(curproc)->b_nsig : NSIG;
1697        caddr_t cmaddr = (caddr_t)arg;
1698        proc_t *p;
1699        user_t *up;
1700        kthread_t *t;
1701        klwp_t *lwp;
1702        prnode_t *pnp = VTOP(vp);
1703        prcommon_t *pcp;
1704        prnode_t *xpnp = NULL;
1705        int error;
1706        int zdisp;
1707        void *thing = NULL;
1708        size_t thingsize = 0;

1710        /*
1711         * For copyin()/copyout().
1712         */
1713        union {
1714                caddr32_t       va;
1715                int             signo;
1716                int             nice;
1717                uint_t          lwpid;
1718                int32_t         flags;
1719                prstatus32_t    prstat;
1720                prrun32_t       prrun;
1721                sigset_t        smask;
1722                siginfo32_t     info;
1723                sysset_t        prmask;
```

```
1724                    prgregset32_t   regs;
1725                    prfpregset32_t  fpregs;
1726                    prpsinfo32_t    prps;
1727                    sigset_t        holdmask;
1728                    fltset_t        fltmask;
1729                    prcred_t        prcred;
1730                    prusage32_t     prusage;
1731                    prhusage_t      prhusage;
1732                    ioc_prmap32_t   prmap;
1733                    auxv32_t        auxv[__KERN_NAUXV_IMPL];
1734            } un32;

1736            /*
1737             * Native objects for internal use.
1738             */
1739            union {
1740                    caddr_t         va;
1741                    int             signo;
1742                    int             nice;
1743                    uint_t          lwpid;
1744                    long            flags;
1745                    prstatus_t      prstat;
1746                    prrun_t         prrun;
1747                    sigset_t        smask;
1748                    siginfo_t       info;
1749                    sysset_t        prmask;
1750                    prgregset_t     regs;
1751                    prpsinfo_t      prps;
1752                    sigset_t        holdmask;
1753                    fltset_t        fltmask;
1754                    prcred_t        prcred;
1755                    prusage_t       prusage;
1756                    prhusage_t      prhusage;
1757                    auxv_t          auxv[__KERN_NAUXV_IMPL];
1758            } un;

1760            if (pnp->pr_type == PR_TMPL)
1761                    return (prctioctl(pnp, cmd, arg, flag, cr));

1763            /*
1764             * Support for old /proc interface.
1765             */
1766            if (pnp->pr_pidfile != NULL) {
1767                    ASSERT(pnp->pr_type == PR_PIDDIR);
1768                    vp = pnp->pr_pidfile;
1769                    pnp = VTOP(vp);
1770                    ASSERT(pnp->pr_type == PR_PIDFILE);
1771            }

1773            if (pnp->pr_type != PR_PIDFILE && pnp->pr_type != PR_LWPIDFILE)
1774                    return (ENOTTY);

1776            /*
1777             * Fail ioctls which are logically "write" requests unless
1778             * the user has write permission.
1779             */
1780            if ((flag & FWRITE) == 0 && isprwrioctl(cmd))
1781                    return (EBADF);

1783            /*
1784             * Perform any necessary copyin() operations before
1785             * locking the process.  Helps avoid deadlocks and
1786             * improves performance.
1787             *
1788             * Also, detect invalid ioctl codes here to avoid
1789             * locking a process unnnecessarily.
```



```
1790             *
1791             * Also, prepare to allocate space that will be needed below,
1792             * case by case.
1793             */
1794            error = 0;
1795            switch (cmd) {
1796            case PIOCGETPR:
1797                    thingsize = sizeof (proc_t);
1798                    break;
1799            case PIOCGETU:
1800                    thingsize = sizeof (user_t);
1801                    break;
1802            case PIOCSTOP:
1803            case PIOCWSTOP:
1804            case PIOCLWPIDS:
1805            case PIOCGTRACE:
1806            case PIOCGENTRY:
1807            case PIOCGEXIT:
1808            case PIOCSRLC:
1809            case PIOCRRLC:
1810            case PIOCSFORK:
1811            case PIOCRFORK:
1812            case PIOCGREG:
1813            case PIOCGFPREG:
1814            case PIOCSTATUS:
1815            case PIOCLSTATUS:
1816            case PIOCPSINFO:
1817            case PIOCMAXSIG:
1818            case PIOCGXREGSIZE:
1819                    break;
1820            case PIOCSXREG:         /* set extra registers */
1821            case PIOCGXREG:         /* get extra registers */
1822 #if defined(__sparc)
1823                    thingsize = sizeof (prxregset_t);
1824 #else
1825                    thingsize = 0;
1826 #endif
1827                    break;
1828            case PIOCACTION:
1829                    thingsize = (nsig-1) * sizeof (struct sigaction32);
1830                    break;
1831            case PIOCGHOLD:
1832            case PIOCNMAP:
1833            case PIOCMAP:
1834            case PIOCGFAULT:
1835            case PIOCCFAULT:
1836            case PIOCCRED:
1837            case PIOCGROUPS:
1838            case PIOCUSAGE:
1839            case PIOCLUSAGE:
1840                    break;
1841            case PIOCOPENPD:
1842                    /*
1843                     * We will need this below.
1844                     * Allocate it now, before locking the process.
1845                     */
1846                    xpnp = prgetnode(vp, PR_OPAGEDATA);
1847                    break;
1848            case PIOCNAUXV:
1849            case PIOCAUXV:
1850                    break;

1852 #if defined(__i386) || defined(__i386_COMPAT)
1853            case PIOCNLDT:
1854            case PIOCLDT:
1855                    break;
```

```
1856 #endif  /* __i386 || __i386_COMPAT */

1858 #if defined(__sparc)
1859         case PIOCGWIN:
1860                 thingsize = sizeof (gwindows32_t);
1861                 break;
1862 #endif  /* __sparc */

1864         case PIOCOPENM:         /* open mapped object for reading */
1865                 if (cmaddr == NULL)
1866                         un32.va = NULL;
1867                 else if (copyin(cmaddr, &un32.va, sizeof (un32.va)))
1868                         error = EFAULT;
1869                 break;

1871         case PIOCRUN:           /* make lwp or process runnable */
1872                 if (cmaddr == NULL)
1873                         un32.prrun.pr_flags = 0;
1874                 else if (copyin(cmaddr, &un32.prrun, sizeof (un32.prrun)))
1875                         error = EFAULT;
1876                 break;

1878         case PIOCOPENLWP:       /* return /proc lwp file descriptor */
1879                 if (copyin(cmaddr, &un32.lwpid, sizeof (un32.lwpid)))
1880                         error = EFAULT;
1881                 break;

1883         case PIOCSTRACE:        /* set signal trace mask */
1884                 if (copyin(cmaddr, &un32.smask, sizeof (un32.smask)))
1885                         error = EFAULT;
1886                 break;

1888         case PIOCSSIG:          /* set current signal */
1889                 if (cmaddr == NULL)
1890                         un32.info.si_signo = 0;
1891                 else if (copyin(cmaddr, &un32.info, sizeof (un32.info)))
1892                         error = EFAULT;
1893                 break;

1895         case PIOCKILL:          /* send signal */
1896         case PIOCUNKILL:        /* delete a signal */
1897                 if (copyin(cmaddr, &un32.signo, sizeof (un32.signo)))
1898                         error = EFAULT;
1899                 break;

1901         case PIOCNICE:          /* set nice priority */
1902                 if (copyin(cmaddr, &un32.nice, sizeof (un32.nice)))
1903                         error = EFAULT;
1904                 break;

1906         case PIOCSENTRY:        /* set syscall entry bit mask */
1907         case PIOCSEXIT:         /* set syscall exit bit mask */
1908                 if (copyin(cmaddr, &un32.prmask, sizeof (un32.prmask)))
1909                         error = EFAULT;
1910                 break;

1912         case PIOCSET:           /* set process flags */
1913         case PIOCRESET:         /* reset process flags */
1914                 if (copyin(cmaddr, &un32.flags, sizeof (un32.flags)))
1915                         error = EFAULT;
1916                 break;

1918         case PIOCSREG:          /* set general registers */
1919                 if (copyin(cmaddr, un32.regs, sizeof (un32.regs)))
1920                         error = EFAULT;
1921                 break;
```



```
1923         case PIOCSFPREG:        /* set floating-point registers */
1924                 if (copyin(cmaddr, &un32.fpregs, sizeof (un32.fpregs)))
1925                         error = EFAULT;
1926                 break;

1928         case PIOCSHOLD:         /* set signal-hold mask */
1929                 if (copyin(cmaddr, &un32.holdmask, sizeof (un32.holdmask)))
1930                         error = EFAULT;
1931                 break;

1933         case PIOCSFAULT:        /* set mask of traced faults */
1934                 if (copyin(cmaddr, &un32.fltmask, sizeof (un32.fltmask)))
1935                         error = EFAULT;
1936                 break;

1938         default:
1939                 error = EINVAL;
1940                 break;
1941         }

1943         if (error)
1944                 return (error);

1946 startover:
1947         /*
1948          * If we need kmem_alloc()d space then we allocate it now, before
1949          * grabbing the process lock.  Using kmem_alloc(KM_SLEEP) while
1950          * holding the process lock leads to deadlock with the clock thread.
1951          * (The clock thread wakes up the pageout daemon to free up space.
1952          * If the clock thread blocks behind us and we are sleeping waiting
1953          * for space, then space may never become available.)
1954          */
1955         if (thingsize) {
1956                 ASSERT(thing == NULL);
1957                 thing = kmem_alloc(thingsize, KM_SLEEP);
1958         }

1960         switch (cmd) {
1961         case PIOCPSINFO:
1962         case PIOCGETPR:
1963         case PIOCUSAGE:
1964         case PIOCLUSAGE:
1965                 zdisp = ZYES;
1966                 break;
1967         case PIOCSXREG:         /* set extra registers */
1968                 /*
1969                  * perform copyin before grabbing the process lock
1970                  */
1971                 if (thing) {
1972                         if (copyin(cmaddr, thing, thingsize)) {
1973                                 kmem_free(thing, thingsize);
1974                                 return (EFAULT);
1975                         }
1976                 }
1977                 /* fall through... */
1978         default:
1979                 zdisp = ZNO;
1980                 break;
1981         }

1983         if ((error = prlock(pnp, zdisp)) != 0) {
1984                 if (thing != NULL)
1985                         kmem_free(thing, thingsize);
1986                 if (xpnp)
1987                         prfreenode(xpnp);
```

```
1988                    return (error);
1989            }

1991            pcp = pnp->pr_common;
1992            p = pcp->prc_proc;
1993            ASSERT(p != NULL);

1995            /*
1996             * Choose a thread/lwp for the operation.
1997             */
1998            if (zdisp == ZNO && cmd != PIOCSTOP && cmd != PIOCWSTOP) {
1999                    if (pnp->pr_type == PR_LWPIDFILE && cmd != PIOCLSTATUS) {
2000                            t = pcp->prc_thread;
2001                            ASSERT(t != NULL);
2002                    } else {
2003                            t = prchoose(p);        /* returns locked thread */
2004                            ASSERT(t != NULL);
2005                            thread_unlock(t);
2006                    }
2007                    lwp = ttolwp(t);
2008            }

2010            error = 0;
2011            switch (cmd) {

2013            case PIOCGETPR:         /* read struct proc */
2014            {
2015                    proc_t *prp = thing;

2017                    *prp = *p;
2018                    prunlock(pnp);
2019                    if (copyout(prp, cmaddr, sizeof (proc_t)))
2020                            error = EFAULT;
2021                    kmem_free(prp, sizeof (proc_t));
2022                    thing = NULL;
2023                    break;
2024            }

2026            case PIOCGETU:          /* read u-area */
2027            {
2028                    user_t *userp = thing;

2030                    up = PTOU(p);
2031                    *userp = *up;
2032                    prunlock(pnp);
2033                    if (copyout(userp, cmaddr, sizeof (user_t)))
2034                            error = EFAULT;
2035                    kmem_free(userp, sizeof (user_t));
2036                    thing = NULL;
2037                    break;
2038            }

2040            case PIOCOPENM:         /* open mapped object for reading */
2041                    if (PROCESS_NOT_32BIT(p) && cmaddr != NULL) {
2042                            prunlock(pnp);
2043                            error = EOVERFLOW;
2044                            break;
2045                    }
2046                    error = propenm(pnp, cmaddr,
2047                        (caddr_t)(uintptr_t)un32.va, rvalp, cr);
2048                    /* propenm() called prunlock(pnp) */
2049                    break;

2051            case PIOCSTOP:          /* stop process or lwp from running */
2052            case PIOCWSTOP:         /* wait for process or lwp to stop */
2053                    /*
```



```
2054                     * Can't apply to a system process.
2055                     */
2056                    if ((p->p_flag & SSYS) || p->p_as == &kas) {
2057                            prunlock(pnp);
2058                            error = EBUSY;
2059                            break;
2060                    }

2062                    if (cmd == PIOCSTOP)
2063                            pr_stop(pnp);

2065                    /*
2066                     * If an lwp is waiting for itself or its process, don't wait.
2067                     * The lwp will never see the fact that itself is stopped.
2068                     */
2069                    if ((pnp->pr_type == PR_LWPIDFILE)?
2070                        (pcp->prc_thread == curthread) : (p == curproc)) {
2071                            if (cmd == PIOCWSTOP)
2072                                    error = EBUSY;
2073                            prunlock(pnp);
2074                            break;
2075                    }

2077                    if ((error = pr_wait_stop(pnp, (time_t)0)) != 0)
2078                            break;  /* pr_wait_stop() unlocked the process */

2080                    if (cmaddr == NULL)
2081                            prunlock(pnp);
2082                    else if (PROCESS_NOT_32BIT(p)) {
2083                            prunlock(pnp);
2084                            error = EOVERFLOW;
2085                    } else {
2086                            /*
2087                             * Return process/lwp status information.
2088                             */
2089                            t = pr_thread(pnp);     /* returns locked thread */
2090                            thread_unlock(t);
2091                            oprgetstatus32(t, &un32.prstat, VTOZONE(vp));
2092                            prunlock(pnp);
2093                            if (copyout(&un32.prstat, cmaddr, sizeof (un32.prstat)))
2094                                    error = EFAULT;
2095                    }
2096                    break;

2098            case PIOCRUN:           /* make lwp or process runnable */
2099            {
2100                    long flags = un32.prrun.pr_flags;

2102                    /*
2103                     * Cannot set an lwp running is it is not stopped.
2104                     * Also, no lwp other than the /proc agent lwp can
2105                     * be set running so long as the /proc agent lwp exists.
2106                     */
2107                    if ((!ISTOPPED(t) && !VSTOPPED(t) &&
2108                        !(t->t_proc_flag & TP_PRSTOP)) ||
2109                        (p->p_agenttp != NULL &&
2110                        (t != p->p_agenttp || pnp->pr_type != PR_LWPIDFILE))) {
2111                            prunlock(pnp);
2112                            error = EBUSY;
2113                            break;
2114                    }

2116                    if ((flags & PRSVADDR) && PROCESS_NOT_32BIT(p)) {
2117                            prunlock(pnp);
2118                            error = EOVERFLOW;
2119                            break;
```

```
2120                }

2122                if (flags & (PRSHOLD|PRSTRACE|PRSFAULT|PRSVADDR)) {
2123                        un.prrun.pr_flags = (int)flags;
2124                        un.prrun.pr_trace = un32.prrun.pr_trace;
2125                        un.prrun.pr_sighold = un32.prrun.pr_sighold;
2126                        un.prrun.pr_fault = un32.prrun.pr_fault;
2127                        un.prrun.pr_vaddr =
2128                            (caddr_t)(uintptr_t)un32.prrun.pr_vaddr;
2129                        prsetrun(t, &un.prrun);
2130                }

2132                error = pr_setrun(pnp, prmaprunflags(flags));

2134                prunlock(pnp);
2135                break;
2136        }

2138        case PIOCLWPIDS:        /* get array of lwp identifiers */
2139        {
2140                int nlwp;
2141                int Nlwp;
2142                id_t *idp;
2143                id_t *Bidp;

2145                Nlwp = nlwp = p->p_lwpcnt;

2147                if (thing && thingsize != (Nlwp+1) * sizeof (id_t)) {
2148                        kmem_free(thing, thingsize);
2149                        thing = NULL;
2150                }
2151                if (thing == NULL) {
2152                        thingsize = (Nlwp+1) * sizeof (id_t);
2153                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
2154                }
2155                if (thing == NULL) {
2156                        prunlock(pnp);
2157                        goto startover;
2158                }

2160                idp = thing;
2161                thing = NULL;
2162                Bidp = idp;
2163                if ((t = p->p_tlist) != NULL) {
2164                        do {
2165                                ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
2166                                ASSERT(nlwp > 0);
2167                                --nlwp;
2168                                *idp++ = t->t_tid;
2169                        } while ((t = t->t_forw) != p->p_tlist);
2170                }
2171                *idp = 0;
2172                ASSERT(nlwp == 0);
2173                prunlock(pnp);
2174                if (copyout(Bidp, cmaddr, (Nlwp+1) * sizeof (id_t)))
2175                        error = EFAULT;
2176                kmem_free(Bidp, (Nlwp+1) * sizeof (id_t));
2177                break;
2178        }

2180        case PIOCOPENLWP:       /* return /proc lwp file descriptor */
2181        {
2182                vnode_t *xvp;
2183                int n;

2185                prunlock(pnp);
```



```
2186                if ((xvp = prlwpnode(pnp, un32.lwpid)) == NULL)
2187                        error = ENOENT;
2188                else if (error = fassign(&xvp, flag & (FREAD|FWRITE), &n)) {
2189                        VN_RELE(xvp);
2190                } else
2191                        *rvalp = n;
2192                break;
2193        }

2195        case PIOCOPENPD:        /* return /proc page data file descriptor */
2196        {
2197                vnode_t *xvp = PTOV(xpnp);
2198                vnode_t *dp = pnp->pr_parent;
2199                int n;

2201                if (PROCESS_NOT_32BIT(p)) {
2202                        prunlock(pnp);
2203                        prfreenode(xpnp);
2204                        xpnp = NULL;
2205                        error = EOVERFLOW;
2206                        break;
2207                }

2209                if (pnp->pr_type == PR_LWPIDFILE) {
2210                        dp = VTOP(dp)->pr_parent;
2211                        dp = VTOP(dp)->pr_parent;
2212                }
2213                ASSERT(VTOP(dp)->pr_type == PR_PIDDIR);

2215                VN_HOLD(dp);
2216                pcp = pnp->pr_pcommon;
2217                xpnp->pr_ino = ptoi(pcp->prc_pid);
2218                xpnp->pr_common = pcp;
2219                xpnp->pr_pcommon = pcp;
2220                xpnp->pr_parent = dp;

2222                xpnp->pr_next = p->p_plist;
2223                p->p_plist = xvp;

2225                prunlock(pnp);
2226                if (error = fassign(&xvp, FREAD, &n)) {
2227                        VN_RELE(xvp);
2228                } else
2229                        *rvalp = n;

2231                xpnp = NULL;
2232                break;
2233        }

2235        case PIOCGTRACE:        /* get signal trace mask */
2236                prassignset(&un32.smask, &p->p_sigmask);
2237                prunlock(pnp);
2238                if (copyout(&un32.smask, cmaddr, sizeof (un32.smask)))
2239                        error = EFAULT;
2240                break;

2242        case PIOCSTRACE:        /* set signal trace mask */
2243                prdelset(&un32.smask, SIGKILL);
2244                prassignset(&p->p_sigmask, &un32.smask);
2245                if (!sigisempty(&p->p_sigmask))
2246                        p->p_proc_flag |= P_PR_TRACE;
2247                else if (prisempty(&p->p_fltmask)) {
2248                        up = PTOU(p);
2249                        if (up->u_systrap == 0)
2250                                p->p_proc_flag &= ~P_PR_TRACE;
2251                }
```

```
2252                prunlock(pnp);
2253                break;

2255        case PIOCSSIG:          /* set current signal */
2256                if (un32.info.si_signo != 0 && PROCESS_NOT_32BIT(p)) {
2257                        prunlock(pnp);
2258                        error = EOVERFLOW;
2259                } else {
2260                        bzero(&un.info, sizeof (un.info));
2261                        siginfo_32tok(&un32.info, (k_siginfo_t *)&un.info);
2262                        error = pr_setsig(pnp, &un.info);
2263                        prunlock(pnp);
2264                        if (un32.info.si_signo == SIGKILL && error == 0)
2265                                pr_wait_die(pnp);
2266                }
2267                break;

2269        case PIOCKILL:          /* send signal */
2270                error = pr_kill(pnp, un32.signo, cr);
2271                prunlock(pnp);
2272                if (un32.signo == SIGKILL && error == 0)
2273                        pr_wait_die(pnp);
2274                break;

2276        case PIOCUNKILL:        /* delete a signal */
2277                error = pr_unkill(pnp, un32.signo);
2278                prunlock(pnp);
2279                break;

2281        case PIOCNICE:          /* set nice priority */
2282                error = pr_nice(p, un32.nice, cr);
2283                prunlock(pnp);
2284                break;

2286        case PIOCGENTRY:        /* get syscall entry bit mask */
2287        case PIOCGEXIT:         /* get syscall exit bit mask */
2288                up = PTOU(p);
2289                if (cmd == PIOCGENTRY) {
2290                        prassignset(&un32.prmask, &up->u_entrymask);
2291                } else {
2292                        prassignset(&un32.prmask, &up->u_exitmask);
2293                }
2294                prunlock(pnp);
2295                if (copyout(&un32.prmask, cmaddr, sizeof (un32.prmask)))
2296                        error = EFAULT;
2297                break;

2299        case PIOCSENTRY:        /* set syscall entry bit mask */
2300        case PIOCSEXIT:         /* set syscall exit bit mask */
2301                pr_setentryexit(p, &un32.prmask, cmd == PIOCSENTRY);
2302                prunlock(pnp);
2303                break;

2305        case PIOCSRLC:          /* obsolete: set running on last /proc close */
2306                error = pr_set(p, prmapsetflags(PR_RLC));
2307                prunlock(pnp);
2308                break;

2310        case PIOCRRLC:          /* obsolete: reset run-on-last-close flag */
2311                error = pr_unset(p, prmapsetflags(PR_RLC));
2312                prunlock(pnp);
2313                break;

2315        case PIOCSFORK:         /* obsolete: set inherit-on-fork flag */
2316                error = pr_set(p, prmapsetflags(PR_FORK));
2317                prunlock(pnp);
```



```
2318                break;

2320        case PIOCRFORK:         /* obsolete: reset inherit-on-fork flag */
2321                error = pr_unset(p, prmapsetflags(PR_FORK));
2322                prunlock(pnp);
2323                break;

2325        case PIOCSET:           /* set process flags */
2326                error = pr_set(p, prmapsetflags((long)un32.flags));
2327                prunlock(pnp);
2328                break;

2330        case PIOCRESET:         /* reset process flags */
2331                error = pr_unset(p, prmapsetflags((long)un32.flags));
2332                prunlock(pnp);
2333                break;

2335        case PIOCGREG:          /* get general registers */
2336                if (PROCESS_NOT_32BIT(p))
2337                        error = EOVERFLOW;
2338                else if (t->t_state != TS_STOPPED && !VSTOPPED(t))
2339                        bzero(un32.regs, sizeof (un32.regs));
2340                else {
2341                        /* drop p_lock while touching the lwp's stack */
2342                        mutex_exit(&p->p_lock);
2343                        prgetprregs32(lwp, un32.regs);
2344                        mutex_enter(&p->p_lock);
2345                }
2346                prunlock(pnp);
2347                if (error == 0 &&
2348                    copyout(un32.regs, cmaddr, sizeof (un32.regs)))
2349                        error = EFAULT;
2350                break;

2352        case PIOCSREG:          /* set general registers */
2353                if (PROCESS_NOT_32BIT(p))
2354                        error = EOVERFLOW;
2355                else if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
2356                        error = EBUSY;
2357                else {
2358                        /* drop p_lock while touching the lwp's stack */
2359                        mutex_exit(&p->p_lock);
2360                        prgregset_32ton(lwp, un32.regs, un.regs);
2361                        prsetprregs(lwp, un.regs, 0);
2362                        mutex_enter(&p->p_lock);
2363                }
2364                prunlock(pnp);
2365                break;

2367        case PIOCGFPREG:        /* get floating-point registers */
2368                if (!prhasfp())
2369                        error = EINVAL; /* No FP support */
2370                else if (PROCESS_NOT_32BIT(p))
2371                        error = EOVERFLOW;
2372                else if (t->t_state != TS_STOPPED && !VSTOPPED(t))
2373                        bzero(&un32.fpregs, sizeof (un32.fpregs));
2374                else {
2375                        /* drop p_lock while touching the lwp's stack */
2376                        mutex_exit(&p->p_lock);
2377                        prgetprfpregs32(lwp, &un32.fpregs);
2378                        mutex_enter(&p->p_lock);
2379                }
2380                prunlock(pnp);
2381                if (error == 0 &&
2382                    copyout(&un32.fpregs, cmaddr, sizeof (un32.fpregs)))
2383                        error = EFAULT;
```

```
2384                 break;

2386         case PIOCSFPREG:        /* set floating-point registers */
2387                 if (!prhasfp())
2388                         error = EINVAL; /* No FP support */
2389                 else if (PROCESS_NOT_32BIT(p))
2390                         error = EOVERFLOW;
2391                 else if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
2392                         error = EBUSY;
2393                 else {
2394                         /* drop p_lock while touching the lwp's stack */
2395                         mutex_exit(&p->p_lock);
2396                         prsetprfpregs32(lwp, &un32.fpregs);
2397                         mutex_enter(&p->p_lock);
2398                 }
2399                 prunlock(pnp);
2400                 break;

2402         case PIOCGXREGSIZE:     /* get the size of the extra registers */
2403         {
2404                 int xregsize;

2406                 if (prhasx(p)) {
2407                         xregsize = prgetprxregsize(p);
2408                         prunlock(pnp);
2409                         if (copyout(&xregsize, cmaddr, sizeof (xregsize)))
2410                                 error = EFAULT;
2411                 } else {
2412                         prunlock(pnp);
2413                         error = EINVAL; /* No extra register support */
2414                 }
2415                 break;
2416         }

2418         case PIOCGXREG:         /* get extra registers */
2419                 if (PROCESS_NOT_32BIT(p))
2420                         error = EOVERFLOW;
2421                 else if (!prhasx(p))
2422                         error = EINVAL; /* No extra register support */
2423                 else {
2424                         bzero(thing, thingsize);
2425                         if (t->t_state == TS_STOPPED || VSTOPPED(t)) {
2426                                 /* drop p_lock to touch the stack */
2427                                 mutex_exit(&p->p_lock);
2428                                 prgetprxregs(lwp, thing);
2429                                 mutex_enter(&p->p_lock);
2430                         }
2431                 }
2432                 prunlock(pnp);
2433                 if (error == 0 &&
2434                     copyout(thing, cmaddr, thingsize))
2435                         error = EFAULT;
2436                 if (thing) {
2437                         kmem_free(thing, thingsize);
2438                         thing = NULL;
2439                 }
2440                 break;

2442         case PIOCSXREG:         /* set extra registers */
2443                 if (PROCESS_NOT_32BIT(p))
2444                         error = EOVERFLOW;
2445                 else if (!ISTOPPED(t) && !VSTOPPED(t) && !DSTOPPED(t))
2446                         error = EBUSY;
2447                 else if (!prhasx(p))
2448                         error = EINVAL; /* No extra register support */
2449                 else if (thing) {
```



```
2450                         /* drop p_lock while touching the lwp's stack */
2451                         mutex_exit(&p->p_lock);
2452                         prsetprxregs(lwp, thing);
2453                         mutex_enter(&p->p_lock);
2454                 }
2455                 prunlock(pnp);
2456                 if (thing) {
2457                         kmem_free(thing, thingsize);
2458                         thing = NULL;
2459                 }
2460                 break;

2462         case PIOCSTATUS:        /* get process/lwp status */
2463                 if (PROCESS_NOT_32BIT(p)) {
2464                         prunlock(pnp);
2465                         error = EOVERFLOW;
2466                         break;
2467                 }
2468                 oprgetstatus32(t, &un32.prstat, VTOZONE(vp));
2469                 prunlock(pnp);
2470                 if (copyout(&un32.prstat, cmaddr, sizeof (un32.prstat)))
2471                         error = EFAULT;
2472                 break;

2474         case PIOCLSTATUS:       /* get status for process & all lwps */
2475         {
2476                 int Nlwp;
2477                 int nlwp;
2478                 prstatus32_t *Bprsp;
2479                 prstatus32_t *prsp;

2481                 if (PROCESS_NOT_32BIT(p)) {
2482                         prunlock(pnp);
2483                         if (thing) {
2484                                 kmem_free(thing, thingsize);
2485                                 thing = NULL;
2486                         }
2487                         error = EOVERFLOW;
2488                         break;
2489                 }

2491                 nlwp = Nlwp = p->p_lwpcnt;

2493                 if (thing && thingsize != (Nlwp+1) * sizeof (prstatus32_t)) {
2494                         kmem_free(thing, thingsize);
2495                         thing = NULL;
2496                 }
2497                 if (thing == NULL) {
2498                         thingsize = (Nlwp+1) * sizeof (prstatus32_t);
2499                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
2500                 }
2501                 if (thing == NULL) {
2502                         prunlock(pnp);
2503                         goto startover;
2504                 }

2506                 Bprsp = (prstatus32_t *)thing;
2507                 thing = NULL;
2508                 prsp = Bprsp;
2509                 oprgetstatus32(t, prsp, VTOZONE(vp));
2510                 t = p->p_tlist;
2511                 do {
2512                         ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
2513                         ASSERT(nlwp > 0);
2514                         --nlwp;
2515                         oprgetstatus32(t, ++prsp, VTOZONE(vp));
```

```
2516                } while ((t = t->t_forw) != p->p_tlist);
2517                ASSERT(nlwp == 0);
2518                prunlock(pnp);
2519                if (copyout(Bprsp, cmaddr, (Nlwp+1) * sizeof (prstatus32_t)))
2520                        error = EFAULT;

2522                kmem_free(Bprsp, (Nlwp + 1) * sizeof (prstatus32_t));
2523                break;
2524        }

2526        case PIOCPSINFO:        /* get ps(1) information */
2527        {
2528                prpsinfo32_t *psp = &un32.prps;

2530                oprgetpsinfo32(p, psp,
2531                    (pnp->pr_type == PR_LWPIDFILE)? pcp->prc_thread : NULL);

2533                prunlock(pnp);
2534                if (copyout(&un32.prps, cmaddr, sizeof (un32.prps)))
2535                        error = EFAULT;
2536                break;
2537        }

2539        case PIOCMAXSIG:        /* get maximum signal number */
2540        {
2541                int n = nsig-1;

2543                prunlock(pnp);
2544                if (copyout(&n, cmaddr, sizeof (int)))
2545                        error = EFAULT;
2546                break;
2547        }

2549        case PIOCACTION:        /* get signal action structures */
2550        {
2551                uint_t sig;
2552                struct sigaction32 *sap = thing;

2554                if (PROCESS_NOT_32BIT(p))
2555                        error = EOVERFLOW;
2556                else {
2557                        up = PTOU(p);
2558                        for (sig = 1; sig < nsig; sig++)
2559                                prgetaction32(p, up, sig, &sap[sig-1]);
2560                }
2561                prunlock(pnp);
2562                if (error == 0 &&
2563                    copyout(sap, cmaddr, (nsig-1)*sizeof (struct sigaction32)))
2564                        error = EFAULT;
2565                kmem_free(sap, (nsig-1)*sizeof (struct sigaction32));
2566                thing = NULL;
2567                break;
2568        }

2570        case PIOCGHOLD:         /* get signal-hold mask */
2571                schedctl_finish_sigblock(t);
2572                sigktou(&t->t_hold, &un32.holdmask);
2573                prunlock(pnp);
2574                if (copyout(&un32.holdmask, cmaddr, sizeof (un32.holdmask)))
2575                        error = EFAULT;
2576                break;

2578        case PIOCSHOLD:         /* set signal-hold mask */
2579                pr_sethold(pnp, &un32.holdmask);
2580                prunlock(pnp);
2581                break;
```



```
2583        case PIOCNMAP:          /* get number of memory mappings */
2584        {
2585                int n;
2586                struct as *as = p->p_as;

2588                if ((p->p_flag & SSYS) || as == &kas)
2589                        n = 0;
2590                else {
2591                        mutex_exit(&p->p_lock);
2592                        AS_LOCK_ENTER(as, RW_WRITER);
2592                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2593                        n = prnsegs(as, 0);
2594                        AS_LOCK_EXIT(as);
2594                        AS_LOCK_EXIT(as, &as->a_lock);
2595                        mutex_enter(&p->p_lock);
2596                }
2597                prunlock(pnp);
2598                if (copyout(&n, cmaddr, sizeof (int)))
2599                        error = EFAULT;
2600                break;
2601        }

2603        case PIOCMAP:           /* get memory map information */
2604        {
2605                list_t iolhead;
2606                struct as *as = p->p_as;

2608                if ((p->p_flag & SSYS) || as == &kas) {
2609                        error = 0;
2610                        prunlock(pnp);
2611                } else if (PROCESS_NOT_32BIT(p)) {
2612                        error = EOVERFLOW;
2613                        prunlock(pnp);
2614                } else {
2615                        mutex_exit(&p->p_lock);
2616                        AS_LOCK_ENTER(as, RW_WRITER);
2616                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2617                        error = oprgetmap32(p, &iolhead);
2618                        AS_LOCK_EXIT(as);
2618                        AS_LOCK_EXIT(as, &as->a_lock);
2619                        mutex_enter(&p->p_lock);
2620                        prunlock(pnp);

2622                        error = pr_iol_copyout_and_free(&iolhead,
2623                            &cmaddr, error);
2624                }
2625                /*
2626                 * The procfs PIOCMAP ioctl returns an all-zero buffer
2627                 * to indicate the end of the prmap[] array.
2628                 * Append it to whatever has already been copied out.
2629                 */
2630                bzero(&un32.prmap, sizeof (un32.prmap));
2631                if (!error &&
2632                    copyout(&un32.prmap, cmaddr, sizeof (un32.prmap)))
2633                        error = EFAULT;
2634                break;
2635        }

2637        case PIOCGFAULT:        /* get mask of traced faults */
2638                prassignset(&un32.fltmask, &p->p_fltmask);
2639                prunlock(pnp);
2640                if (copyout(&un32.fltmask, cmaddr, sizeof (un32.fltmask)))
2641                        error = EFAULT;
2642                break;
```

```
2644         case PIOCSFAULT:        /* set mask of traced faults */
2645                 pr_setfault(p, &un32.fltmask);
2646                 prunlock(pnp);
2647                 break;

2649         case PIOCCFAULT:        /* clear current fault */
2650                 lwp->lwp_curflt = 0;
2651                 prunlock(pnp);
2652                 break;

2654         case PIOCCRED:          /* get process credentials */
2655         {
2656                 cred_t *cp;

2658                 mutex_enter(&p->p_crlock);
2659                 cp = p->p_cred;
2660                 un32.prcred.pr_euid = crgetuid(cp);
2661                 un32.prcred.pr_ruid = crgetruid(cp);
2662                 un32.prcred.pr_suid = crgetsuid(cp);
2663                 un32.prcred.pr_egid = crgetgid(cp);
2664                 un32.prcred.pr_rgid = crgetrgid(cp);
2665                 un32.prcred.pr_sgid = crgetsgid(cp);
2666                 un32.prcred.pr_ngroups = crgetngroups(cp);
2667                 mutex_exit(&p->p_crlock);

2669                 prunlock(pnp);
2670                 if (copyout(&un32.prcred, cmaddr, sizeof (un32.prcred)))
2671                         error = EFAULT;
2672                 break;
2673         }

2675         case PIOCGROUPS:        /* get supplementary groups */
2676         {
2677                 cred_t *cp;

2679                 mutex_enter(&p->p_crlock);
2680                 cp = p->p_cred;
2681                 crhold(cp);
2682                 mutex_exit(&p->p_crlock);

2684                 prunlock(pnp);
2685                 if (copyout(crgetgroups(cp), cmaddr,
2686                     MAX(crgetngroups(cp), 1) * sizeof (gid_t)))
2687                         error = EFAULT;
2688                 crfree(cp);
2689                 break;
2690         }

2692         case PIOCUSAGE:         /* get usage info */
2693         {
2694                 /*
2695                  * For an lwp file descriptor, return just the lwp usage.
2696                  * For a process file descriptor, return total usage,
2697                  * all current lwps plus all defunct lwps.
2698                  */
2699                 prhusage_t *pup = &un32.prhusage;
2700                 prusage32_t *upup;

2702                 bzero(pup, sizeof (*pup));
2703                 pup->pr_tstamp = gethrtime();

2705                 if (pnp->pr_type == PR_LWPIDFILE) {
2706                         t = pcp->prc_thread;
2707                         if (t != NULL)
2708                                 prgetusage(t, pup);
2709                         else
```



```
2710                                 error = ENOENT;
2711                 } else {
2712                         pup->pr_count  = p->p_defunct;
2713                         pup->pr_create = p->p_mstart;
2714                         pup->pr_term   = p->p_mterm;

2716                         pup->pr_rtime    = p->p_mlreal;
2717                         pup->pr_utime    = p->p_acct[LMS_USER];
2718                         pup->pr_stime    = p->p_acct[LMS_SYSTEM];
2719                         pup->pr_ttime    = p->p_acct[LMS_TRAP];
2720                         pup->pr_tftime   = p->p_acct[LMS_TFAULT];
2721                         pup->pr_dftime   = p->p_acct[LMS_DFAULT];
2722                         pup->pr_kftime   = p->p_acct[LMS_KFAULT];
2723                         pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
2724                         pup->pr_slptime  = p->p_acct[LMS_SLEEP];
2725                         pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
2726                         pup->pr_stoptime = p->p_acct[LMS_STOPPED];

2728                         pup->pr_minf  = p->p_ru.minflt;
2729                         pup->pr_majf  = p->p_ru.majflt;
2730                         pup->pr_nswap = p->p_ru.nswap;
2731                         pup->pr_inblk = p->p_ru.inblock;
2732                         pup->pr_oublk = p->p_ru.oublock;
2733                         pup->pr_msnd  = p->p_ru.msgsnd;
2734                         pup->pr_mrcv  = p->p_ru.msgrcv;
2735                         pup->pr_sigs  = p->p_ru.nsignals;
2736                         pup->pr_vctx  = p->p_ru.nvcsw;
2737                         pup->pr_ictx  = p->p_ru.nivcsw;
2738                         pup->pr_sysc  = p->p_ru.sysc;
2739                         pup->pr_ioch  = p->p_ru.ioch;

2741                         /*
2742                          * Add the usage information for each active lwp.
2743                          */
2744                         if ((t = p->p_tlist) != NULL &&
2745                             !(pcp->prc_flags & PRC_DESTROY)) {
2746                                 do {
2747                                         ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
2748                                         pup->pr_count++;
2749                                         praddusage(t, pup);
2750                                 } while ((t = t->t_forw) != p->p_tlist);
2751                         }
2752                 }

2754                 prunlock(pnp);

2756                 upup = kmem_alloc(sizeof (*upup), KM_SLEEP);
2757                 prcvtusage32(pup, upup);
2758                 if (copyout(upup, cmaddr, sizeof (*upup)))
2759                         error = EFAULT;
2760                 kmem_free(upup, sizeof (*upup));

2762                 break;
2763         }

2765         case PIOCLUSAGE:        /* get detailed usage info */
2766         {
2767                 int Nlwp;
2768                 int nlwp;
2769                 prusage32_t *upup;
2770                 prusage32_t *Bupup;
2771                 prhusage_t *pup;
2772                 hrtime_t curtime;

2774                 nlwp = Nlwp = (pcp->prc_flags & PRC_DESTROY)? 0 : p->p_lwpcnt;
```

```
2776                if (thing && thingsize !=
2777                    sizeof (prhusage_t) + (Nlwp+1) * sizeof (prusage32_t)) {
2778                        kmem_free(thing, thingsize);
2779                        thing = NULL;
2780                }
2781                if (thing == NULL) {
2782                        thingsize = sizeof (prhusage_t) +
2783                            (Nlwp+1) * sizeof (prusage32_t);
2784                        thing = kmem_alloc(thingsize, KM_NOSLEEP);
2785                }
2786                if (thing == NULL) {
2787                        prunlock(pnp);
2788                        goto startover;
2789                }

2791                pup = (prhusage_t *)thing;
2792                upup = Bupup = (prusage32_t *)(pup + 1);

2794                ASSERT(p == pcp->prc_proc);

2796                curtime = gethrtime();

2798                /*
2799                 * First the summation over defunct lwps.
2800                 */
2801                bzero(pup, sizeof (*pup));
2802                pup->pr_count  = p->p_defunct;
2803                pup->pr_tstamp = curtime;
2804                pup->pr_create = p->p_mstart;
2805                pup->pr_term   = p->p_mterm;

2807                pup->pr_rtime    = p->p_mlreal;
2808                pup->pr_utime    = p->p_acct[LMS_USER];
2809                pup->pr_stime    = p->p_acct[LMS_SYSTEM];
2810                pup->pr_ttime    = p->p_acct[LMS_TRAP];
2811                pup->pr_tftime   = p->p_acct[LMS_TFAULT];
2812                pup->pr_dftime   = p->p_acct[LMS_DFAULT];
2813                pup->pr_kftime   = p->p_acct[LMS_KFAULT];
2814                pup->pr_ltime    = p->p_acct[LMS_USER_LOCK];
2815                pup->pr_slptime  = p->p_acct[LMS_SLEEP];
2816                pup->pr_wtime    = p->p_acct[LMS_WAIT_CPU];
2817                pup->pr_stoptime = p->p_acct[LMS_STOPPED];

2819                pup->pr_minf  = p->p_ru.minflt;
2820                pup->pr_majf  = p->p_ru.majflt;
2821                pup->pr_nswap = p->p_ru.nswap;
2822                pup->pr_inblk = p->p_ru.inblock;
2823                pup->pr_oublk = p->p_ru.oublock;
2824                pup->pr_msnd  = p->p_ru.msgsnd;
2825                pup->pr_mrcv  = p->p_ru.msgrcv;
2826                pup->pr_sigs  = p->p_ru.nsignals;
2827                pup->pr_vctx  = p->p_ru.nvcsw;
2828                pup->pr_ictx  = p->p_ru.nivcsw;
2829                pup->pr_sysc  = p->p_ru.sysc;
2830                pup->pr_ioch  = p->p_ru.ioch;

2832                prcvtusage32(pup, upup);

2834                /*
2835                 * Fill one prusage struct for each active lwp.
2836                 */
2837                if ((t = p->p_tlist) != NULL &&
2838                    !(pcp->prc_flags & PRC_DESTROY)) {
2839                        do {
2840                                ASSERT(!(t->t_proc_flag & TP_LWPEXIT));
2841                                ASSERT(nlwp > 0);
```



```
2842                                --nlwp;
2843                                upup++;
2844                                prgetusage(t, pup);
2845                                prcvtusage32(pup, upup);
2846                        } while ((t = t->t_forw) != p->p_tlist);
2847                }
2848                ASSERT(nlwp == 0);

2850                prunlock(pnp);
2851                if (copyout(Bupup, cmaddr, (Nlwp+1) * sizeof (prusage32_t)))
2852                        error = EFAULT;
2853                kmem_free(thing, thingsize);
2854                thing = NULL;
2855                break;
2856        }

2858        case PIOCNAUXV:         /* get number of aux vector entries */
2859        {
2860                int n = __KERN_NAUXV_IMPL;

2862                prunlock(pnp);
2863                if (copyout(&n, cmaddr, sizeof (int)))
2864                        error = EFAULT;
2865                break;
2866        }

2868        case PIOCAUXV:          /* get aux vector (see sys/auxv.h) */
2869        {
2870                int i;

2872                if (PROCESS_NOT_32BIT(p)) {
2873                        prunlock(pnp);
2874                        error = EOVERFLOW;
2875                } else {
2876                        up = PTOU(p);
2877                        for (i = 0; i < __KERN_NAUXV_IMPL; i++) {
2878                                un32.auxv[i].a_type = up->u_auxv[i].a_type;
2879                                un32.auxv[i].a_un.a_val =
2880                                    (int32_t)up->u_auxv[i].a_un.a_val;
2881                        }
2882                        prunlock(pnp);
2883                        if (copyout(un32.auxv, cmaddr,
2884                            __KERN_NAUXV_IMPL * sizeof (auxv32_t)))
2885                                error = EFAULT;
2886                }
2887                break;
2888        }

2890 #if defined(__i386) || defined(__i386_COMPAT)
2891        case PIOCNLDT:          /* get number of LDT entries */
2892        {
2893                int n;

2895                mutex_exit(&p->p_lock);
2896                mutex_enter(&p->p_ldtlock);
2897                n = prnldt(p);
2898                mutex_exit(&p->p_ldtlock);
2899                mutex_enter(&p->p_lock);
2900                prunlock(pnp);
2901                if (copyout(&n, cmaddr, sizeof (n)))
2902                        error = EFAULT;
2903                break;
2904        }

2906        case PIOCLDT:           /* get LDT entries */
2907        {
```

```
2908                 struct ssd *ssd;
2909                 int n;

2911                 mutex_exit(&p->p_lock);
2912                 mutex_enter(&p->p_ldtlock);
2913                 n = prnldt(p);

2915                 if (thing && thingsize != (n+1) * sizeof (*ssd)) {
2916                         kmem_free(thing, thingsize);
2917                         thing = NULL;
2918                 }
2919                 if (thing == NULL) {
2920                         thingsize = (n+1) * sizeof (*ssd);
2921                         thing = kmem_alloc(thingsize, KM_NOSLEEP);
2922                 }
2923                 if (thing == NULL) {
2924                         mutex_exit(&p->p_ldtlock);
2925                         mutex_enter(&p->p_lock);
2926                         prunlock(pnp);
2927                         goto startover;
2928                 }

2930                 ssd = thing;
2931                 thing = NULL;
2932                 if (n != 0)
2933                         prgetldt(p, ssd);
2934                 mutex_exit(&p->p_ldtlock);
2935                 mutex_enter(&p->p_lock);
2936                 prunlock(pnp);

2938                 /* mark the end of the list with a null entry */
2939                 bzero(&ssd[n], sizeof (*ssd));
2940                 if (copyout(ssd, cmaddr, (n+1) * sizeof (*ssd)))
2941                         error = EFAULT;
2942                 kmem_free(ssd, (n+1) * sizeof (*ssd));
2943                 break;
2944         }
2945 #endif  /* __i386 || __i386_COMPAT */

2947 #if defined(__sparc)
2948         case PIOCGWIN:          /* get gwindows_t (see sys/reg.h) */
2949         {
2950                 gwindows32_t *gwp = thing;

2952                 if (PROCESS_NOT_32BIT(p)) {
2953                         prunlock(pnp);
2954                         error = EOVERFLOW;
2955                 } else {
2956                         /* drop p->p_lock while touching the stack */
2957                         mutex_exit(&p->p_lock);
2958                         bzero(gwp, sizeof (*gwp));
2959                         prgetwindows32(lwp, gwp);
2960                         mutex_enter(&p->p_lock);
2961                         prunlock(pnp);
2962                         if (copyout(gwp, cmaddr, sizeof (*gwp)))
2963                                 error = EFAULT;
2964                 }
2965                 kmem_free(gwp, sizeof (*gwp));
2966                 thing = NULL;
2967                 break;
2968         }
2969 #endif  /* __sparc */

2971         default:
2972                 prunlock(pnp);
2973                 error = EINVAL;
```



```
2974                 break;

2976         }

2978         ASSERT(thing == NULL);
2979         ASSERT(xpnp == NULL);
2980         return (error);
2981 }
_____unchanged_portion_omitted_

3117 /*
3118  * Common code for PIOCOPENM
3119  * Returns with the process unlocked.
3120  */
3121 static int
3122 propenm(prnode_t *pnp, caddr_t cmaddr, caddr_t va, int *rvalp, cred_t *cr)
3123 {
3124         proc_t *p = pnp->pr_common->prc_proc;
3125         struct as *as = p->p_as;
3126         int error = 0;
3127         struct seg *seg;
3128         struct vnode *xvp;
3129         int n;

3131         /*
3132          * By fiat, a system process has no address space.
3133          */
3134         if ((p->p_flag & SSYS) || as == &kas) {
3135                 error = EINVAL;
3136         } else if (cmaddr) {
3137                 /*
3138                  * We drop p_lock before grabbing the address
3139                  * space lock in order to avoid a deadlock with
3140                  * the clock thread.  The process will not
3141                  * disappear and its address space will not
3142                  * change because it is marked P_PR_LOCK.
3143                  */
3144                 mutex_exit(&p->p_lock);
3145                 AS_LOCK_ENTER(as, RW_READER);
3145                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3146                 seg = as_segat(as, va);
3147                 if (seg != NULL &&
3148                     seg->s_ops == &segvn_ops &&
3149                     SEGOP_GETVP(seg, va, &xvp) == 0 &&
3150                     xvp != NULL &&
3151                     xvp->v_type == VREG) {
3152                         VN_HOLD(xvp);
3153                 } else {
3154                         error = EINVAL;
3155                 }
3156                 AS_LOCK_EXIT(as);
3156                 AS_LOCK_EXIT(as, &as->a_lock);
3157                 mutex_enter(&p->p_lock);
3158         } else if ((xvp = p->p_exec) == NULL) {
3159                 error = EINVAL;
3160         } else {
3161                 VN_HOLD(xvp);
3162         }

3164         prunlock(pnp);

3166         if (error == 0) {
3167                 if ((error = VOP_ACCESS(xvp, VREAD, 0, cr, NULL)) == 0)
3168                         error = fassign(&xvp, FREAD, &n);
3169                 if (error) {
3170                         VN_RELE(xvp);
```

```
3171                } else {
3172                        *rvalp = n;
3173                }
3174        }

3176        return (error);
3177 }
_____unchanged_portion_omitted_

3329 /*
3330  * Return old version of information used by ps(1).
3331  */
3332 void
3333 oprgetpsinfo(proc_t *p, prpsinfo_t *psp, kthread_t *tp)
3334 {
3335        kthread_t *t;
3336        char c, state;
3337        user_t *up;
3338        dev_t d;
3339        uint64_t pct;
3340        int retval, niceval;
3341        cred_t *cred;
3342        struct as *as;
3343        hrtime_t hrutime, hrstime, cur_time;

3345        ASSERT(MUTEX_HELD(&p->p_lock));

3347        bzero(psp, sizeof (*psp));

3349        if ((t = tp) == NULL)
3350                t = prchoose(p);        /* returns locked thread */
3351        else
3352                thread_lock(t);

3354        /* kludge: map thread state enum into process state enum */

3356        if (t == NULL) {
3357                state = TS_ZOMB;
3358        } else {
3359                state = VSTOPPED(t) ? TS_STOPPED : t->t_state;
3360                thread_unlock(t);
3361        }

3363        switch (state) {
3364        case TS_SLEEP:          state = SSLEEP;         break;
3365        case TS_RUN:            state = SRUN;           break;
3366        case TS_ONPROC:         state = SONPROC;        break;
3367        case TS_ZOMB:           state = SZOMB;          break;
3368        case TS_STOPPED:        state = SSTOP;          break;
3369        default:                state = 0;              break;
3370        }
3371        switch (state) {
3372        case SSLEEP:    c = 'S';        break;
3373        case SRUN:      c = 'R';        break;
3374        case SZOMB:     c = 'Z';        break;
3375        case SSTOP:     c = 'T';        break;
3376        case SIDL:      c = 'I';        break;
3377        case SONPROC:   c = 'O';        break;
3378 #ifdef SXBRK
3379        case SXBRK:     c = 'X';        break;
3380 #endif
3381        default:        c = '?';        break;
3382        }
3383        psp->pr_state = state;
3384        psp->pr_sname = c;
3385        psp->pr_zomb = (state == SZOMB);
```



```
3386        /*
3387         * only export SSYS and SMSACCT; everything else is off-limits to
3388         * userland apps.
3389         */
3390        psp->pr_flag = p->p_flag & (SSYS | SMSACCT);

3392        mutex_enter(&p->p_crlock);
3393        cred = p->p_cred;
3394        psp->pr_uid = crgetruid(cred);
3395        psp->pr_gid = crgetrgid(cred);
3396        psp->pr_euid = crgetuid(cred);
3397        psp->pr_egid = crgetgid(cred);
3398        mutex_exit(&p->p_crlock);

3400        psp->pr_pid = p->p_pid;
3401        if (curproc->p_zone->zone_id != GLOBAL_ZONEID &&
3402            (p->p_flag & SZONETOP)) {
3403                ASSERT(p->p_zone->zone_id != GLOBAL_ZONEID);
3404                /*
3405                 * Inside local zones, fake zsched's pid as parent pids for
3406                 * processes which reference processes outside of the zone.
3407                 */
3408                psp->pr_ppid = curproc->p_zone->zone_zsched->p_pid;
3409        } else {
3410                psp->pr_ppid = p->p_ppid;
3411        }
3412        psp->pr_pgrp = p->p_pgrp;
3413        psp->pr_sid = p->p_sessp->s_sid;
3414        psp->pr_addr = prgetpsaddr(p);
3415        hrutime = mstate_aggr_state(p, LMS_USER);
3416        hrstime = mstate_aggr_state(p, LMS_SYSTEM);
3417        hrt2ts(hrutime + hrstime, &psp->pr_time);
3418        TICK_TO_TIMESTRUC(p->p_cutime + p->p_cstime, &psp->pr_ctime);
3419        switch (p->p_model) {
3420        case DATAMODEL_ILP32:
3421                psp->pr_dmodel = PR_MODEL_ILP32;
3422                break;
3423        case DATAMODEL_LP64:
3424                psp->pr_dmodel = PR_MODEL_LP64;
3425                break;
3426        }
3427        if (state == SZOMB || t == NULL) {
3428                int wcode = p->p_wcode;         /* must be atomic read */

3430                if (wcode)
3431                        psp->pr_wstat = wstat(wcode, p->p_wdata);
3432                psp->pr_lttydev = PRNODEV;
3433                psp->pr_ottydev = (o_dev_t)PRNODEV;
3434                psp->pr_size = 0;
3435                psp->pr_rssize = 0;
3436                psp->pr_pctmem = 0;
3437        } else {
3438                up = PTOU(p);
3439                psp->pr_wchan = t->t_wchan;
3440                psp->pr_pri = t->t_pri;
3441                (void) strncpy(psp->pr_clname, sclass[t->t_cid].cl_name,
3442                    sizeof (psp->pr_clname) - 1);
3443                retval = CL_DONICE(t, NULL, 0, &niceval);
3444                if (retval == 0) {
3445                        psp->pr_oldpri = v.v_maxsyspri - psp->pr_pri;
3446                        psp->pr_nice = niceval + NZERO;
3447                } else {
3448                        psp->pr_oldpri = 0;
3449                        psp->pr_nice = 0;
3450                }
3451                d = cttydev(p);
```

```
#ifdef sun
                {
                        extern dev_t rwsconsdev, rconsdev, uconsdev;
                        /*
                         * If the controlling terminal is the real
                         * or workstation console device, map to what the
                         * user thinks is the console device. Handle case when
                         * rwsconsdev or rconsdev is set to NODEV for Starfire.
                         */
                        if ((d == rwsconsdev || d == rconsdev) && d != NODEV)
                                d = uconsdev;
                }
#endif
                psp->pr_lttydev = (d == NODEV) ? PRNODEV : d;
                psp->pr_ottydev = cmpdev(d);
                psp->pr_start = up->u_start;
                bcopy(up->u_comm, psp->pr_fname,
                    MIN(sizeof (up->u_comm), sizeof (psp->pr_fname)-1));
                bcopy(up->u_psargs, psp->pr_psargs,
                    MIN(PRARGSZ-1, PSARGSZ));
                psp->pr_syscall = t->t_sysnum;
                psp->pr_argc = up->u_argc;
                psp->pr_argv = (char **)up->u_argv;
                psp->pr_envp = (char **)up->u_envp;

                /* compute %cpu for the lwp or process */
                pct = 0;
                if ((t = tp) == NULL)
                        t = p->p_tlist;
                cur_time = gethrtime_unscaled();
                do {
                        pct += cpu_update_pct(t, cur_time);
                        if (tp != NULL)         /* just do the one lwp */
                                break;
                } while ((t = t->t_forw) != p->p_tlist);

                psp->pr_pctcpu = prgetpctcpu(pct);
                psp->pr_cpu = (psp->pr_pctcpu*100 + 0x6000) >> 15; /* [0..99] */
                if (psp->pr_cpu > 99)
                        psp->pr_cpu = 99;

                if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
                        psp->pr_size = 0;
                        psp->pr_rssize = 0;
                        psp->pr_pctmem = 0;
                } else {
                        mutex_exit(&p->p_lock);
                        AS_LOCK_ENTER(as, RW_READER);
                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
                        psp->pr_size = btopr(as->a_resvsize);
                        psp->pr_rssize = rm_asrss(as);
                        psp->pr_pctmem = rm_pctmemory(as);
                        AS_LOCK_EXIT(as);
                        AS_LOCK_EXIT(as, &as->a_lock);
                        mutex_enter(&p->p_lock);
                }
        }
        psp->pr_bysize = ptob(psp->pr_size);
        psp->pr_byrssize = ptob(psp->pr_rssize);
}

/*
 * Return an array of structures with memory map information.
 * We allocate here; the caller must deallocate.
 * The caller is also responsible to append the zero-filled entry
 * that terminates the PIOCMAP output buffer.
```



```
 */
static int
oprgetmap(proc_t *p, list_t *iolhead)
{
        struct as *as = p->p_as;
        prmap_t *mp;
        struct seg *seg;
        struct seg *brkseg, *stkseg;
        uint_t prot;

        ASSERT(as != &kas && AS_WRITE_HELD(as));
        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

        /*
         * Request an initial buffer size that doesn't waste memory
         * if the address space has only a small number of segments.
         */
        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

        if ((seg = AS_SEGFIRST(as)) == NULL)
                return (0);

        brkseg = break_seg(p);
        stkseg = as_segat(as, prgetstackbase(p));

        do {
                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
                caddr_t saddr, naddr;
                void *tmp = NULL;

                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
                        prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
                        if (saddr == naddr)
                                continue;

                        mp = pr_iol_newbuf(iolhead, sizeof (*mp));

                        mp->pr_vaddr = saddr;
                        mp->pr_size = naddr - saddr;
                        mp->pr_off = SEGOP_GETOFFSET(seg, saddr);
                        mp->pr_mflags = 0;
                        if (prot & PROT_READ)
                                mp->pr_mflags |= MA_READ;
                        if (prot & PROT_WRITE)
                                mp->pr_mflags |= MA_WRITE;
                        if (prot & PROT_EXEC)
                                mp->pr_mflags |= MA_EXEC;
                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
                                mp->pr_mflags |= MA_SHARED;
                        if (seg == brkseg)
                                mp->pr_mflags |= MA_BREAK;
                        else if (seg == stkseg)
                                mp->pr_mflags |= MA_STACK;
                        mp->pr_pagesize = PAGESIZE;
                }
                ASSERT(tmp == NULL);
        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

        return (0);
}

#ifdef _SYSCALL32_IMPL
static int
oprgetmap32(proc_t *p, list_t *iolhead)
{
        struct as *as = p->p_as;
```

```
3581         ioc_prmap32_t *mp;
3582         struct seg *seg;
3583         struct seg *brkseg, *stkseg;
3584         uint_t prot;

3586         ASSERT(as != &kas && AS_WRITE_HELD(as));
3586         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3588         /*
3589          * Request an initial buffer size that doesn't waste memory
3590          * if the address space has only a small number of segments.
3591          */
3592         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

3594         if ((seg = AS_SEGFIRST(as)) == NULL)
3595                 return (0);

3597         brkseg = break_seg(p);
3598         stkseg = as_segat(as, prgetstackbase(p));

3600         do {
3601                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3602                 caddr_t saddr, naddr;
3603                 void *tmp = NULL;

3605                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3606                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3607                         if (saddr == naddr)
3608                                 continue;

3610                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

3612                         mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
3613                         mp->pr_size = (size32_t)(naddr - saddr);
3614                         mp->pr_off = (off32_t)SEGOP_GETOFFSET(seg, saddr);
3615                         mp->pr_mflags = 0;
3616                         if (prot & PROT_READ)
3617                                 mp->pr_mflags |= MA_READ;
3618                         if (prot & PROT_WRITE)
3619                                 mp->pr_mflags |= MA_WRITE;
3620                         if (prot & PROT_EXEC)
3621                                 mp->pr_mflags |= MA_EXEC;
3622                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3623                                 mp->pr_mflags |= MA_SHARED;
3624                         if (seg == brkseg)
3625                                 mp->pr_mflags |= MA_BREAK;
3626                         else if (seg == stkseg)
3627                                 mp->pr_mflags |= MA_STACK;
3628                         mp->pr_pagesize = PAGESIZE;
3629                 }
3630                 ASSERT(tmp == NULL);
3631         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3633         return (0);
3634 }
3635 #endif  /* _SYSCALL32_IMPL */

3637 /*
3638  * Return the size of the old /proc page data file.
3639  */
3640 size_t
3641 oprpdsize(struct as *as)
3642 {
3643         struct seg *seg;
3644         size_t size;
```



```
3646         ASSERT(as != &kas && AS_WRITE_HELD(as));
3646         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3648         if ((seg = AS_SEGFIRST(as)) == NULL)
3649                 return (0);

3651         size = sizeof (prpageheader_t);
3652         do {
3653                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3654                 caddr_t saddr, naddr;
3655                 void *tmp = NULL;
3656                 size_t npage;

3658                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3659                         (void) pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3660                         if ((npage = (naddr - saddr) / PAGESIZE) != 0)
3661                                 size += sizeof (prasmap_t) + roundlong(npage);
3662                 }
3663                 ASSERT(tmp == NULL);
3664         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3666         return (size);
3667 }

3669 #ifdef _SYSCALL32_IMPL
3670 size_t
3671 oprpdsize32(struct as *as)
3672 {
3673         struct seg *seg;
3674         size_t size;

3676         ASSERT(as != &kas && AS_WRITE_HELD(as));
3676         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3678         if ((seg = AS_SEGFIRST(as)) == NULL)
3679                 return (0);

3681         size = sizeof (ioc_prpageheader32_t);
3682         do {
3683                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3684                 caddr_t saddr, naddr;
3685                 void *tmp = NULL;
3686                 size_t npage;

3688                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3689                         (void) pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3690                         if ((npage = (naddr - saddr) / PAGESIZE) != 0)
3691                                 size += sizeof (ioc_prmap32_t) + round4(npage);
3692                 }
3693                 ASSERT(tmp == NULL);
3694         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3696         return (size);
3697 }
3698 #endif  /* _SYSCALL32_IMPL */

3700 /*
3701  * Read old /proc page data information.
3702  */
3703 int
3704 oprpdread(struct as *as, uint_t hatid, struct uio *uiop)
3705 {
3706         caddr_t buf;
3707         size_t size;
3708         prpageheader_t *php;
3709         prasmap_t *pmp;
```

```
3710         struct seg *seg;
3711         int error;

3713 again:
3714         AS_LOCK_ENTER(as, RW_WRITER);
3714         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3716         if ((seg = AS_SEGFIRST(as)) == NULL) {
3717                 AS_LOCK_EXIT(as);
3717                 AS_LOCK_EXIT(as, &as->a_lock);
3718                 return (0);
3719         }
3720         size = oprpdsize(as);
3721         if (uiop->uio_resid < size) {
3722                 AS_LOCK_EXIT(as);
3722                 AS_LOCK_EXIT(as, &as->a_lock);
3723                 return (E2BIG);
3724         }

3726         buf = kmem_zalloc(size, KM_SLEEP);
3727         php = (prpageheader_t *)buf;
3728         pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));

3730         hrt2ts(gethrtime(), &php->pr_tstamp);
3731         php->pr_nmap = 0;
3732         php->pr_npage = 0;
3733         do {
3734                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3735                 caddr_t saddr, naddr;
3736                 void *tmp = NULL;

3738                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3739                         size_t len;
3740                         size_t npage;
3741                         uint_t prot;
3742                         uintptr_t next;

3744                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3745                         if ((len = naddr - saddr) == 0)
3746                                 continue;
3747                         npage = len / PAGESIZE;
3748                         next = (uintptr_t)(pmp + 1) + roundlong(npage);
3749                         /*
3750                          * It's possible that the address space can change
3751                          * subtlely even though we're holding as->a_lock
3752                          * due to the nondeterminism of page_exists() in
3753                          * the presence of asychronously flushed pages or
3754                          * mapped files whose sizes are changing.
3755                          * page_exists() may be called indirectly from
3756                          * pr_getprot() by a SEGOP_INCORE() routine.
3757                          * If this happens we need to make sure we don't
3758                          * overrun the buffer whose size we computed based
3759                          * on the initial iteration through the segments.
3760                          * Once we've detected an overflow, we need to clean
3761                          * up the temporary memory allocated in pr_getprot()
3762                          * and retry. If there's a pending signal, we return
3763                          * EINTR so that this thread can be dislodged if
3764                          * a latent bug causes us to spin indefinitely.
3765                          */
3766                         if (next > (uintptr_t)buf + size) {
3767                                 pr_getprot_done(&tmp);
3768                                 AS_LOCK_EXIT(as);
3768                                 AS_LOCK_EXIT(as, &as->a_lock);

3770                                 kmem_free(buf, size);
```



```
3772                                 if (ISSIG(curthread, JUSTLOOKING))
3773                                         return (EINTR);

3775                                 goto again;
3776                         }

3778                         php->pr_nmap++;
3779                         php->pr_npage += npage;
3780                         pmp->pr_vaddr = saddr;
3781                         pmp->pr_npage = npage;
3782                         pmp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3783                         pmp->pr_mflags = 0;
3784                         if (prot & PROT_READ)
3785                                 pmp->pr_mflags |= MA_READ;
3786                         if (prot & PROT_WRITE)
3787                                 pmp->pr_mflags |= MA_WRITE;
3788                         if (prot & PROT_EXEC)
3789                                 pmp->pr_mflags |= MA_EXEC;
3790                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3791                                 pmp->pr_mflags |= MA_SHARED;
3792                         pmp->pr_pagesize = PAGESIZE;
3793                         hat_getstat(as, saddr, len, hatid,
3794                             (char *)(pmp + 1), HAT_SYNC_ZERORM);
3795                         pmp = (prasmap_t *)next;
3796                 }
3797                 ASSERT(tmp == NULL);
3798         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3800         AS_LOCK_EXIT(as);
3800         AS_LOCK_EXIT(as, &as->a_lock);

3802         ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
3803         error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3804         kmem_free(buf, size);

3806         return (error);
3807 }

3809 #ifdef _SYSCALL32_IMPL
3810 int
3811 oprpdread32(struct as *as, uint_t hatid, struct uio *uiop)
3812 {
3813         caddr_t buf;
3814         size_t size;
3815         ioc_prpageheader32_t *php;
3816         ioc_prasmap32_t *pmp;
3817         struct seg *seg;
3818         int error;

3820 again:
3821         AS_LOCK_ENTER(as, RW_WRITER);
3821         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3823         if ((seg = AS_SEGFIRST(as)) == NULL) {
3824                 AS_LOCK_EXIT(as);
3824                 AS_LOCK_EXIT(as, &as->a_lock);
3825                 return (0);
3826         }
3827         size = oprpdsize32(as);
3828         if (uiop->uio_resid < size) {
3829                 AS_LOCK_EXIT(as);
3829                 AS_LOCK_EXIT(as, &as->a_lock);
3830                 return (E2BIG);
3831         }

3833         buf = kmem_zalloc(size, KM_SLEEP);
```

```
3834         php = (ioc_prpageheader32_t *)buf;
3835         pmp = (ioc_prasmap32_t *)(buf + sizeof (ioc_prpageheader32_t));

3837         hrt2ts32(gethrtime(), &php->pr_tstamp);
3838         php->pr_nmap = 0;
3839         php->pr_npage = 0;
3840         do {
3841                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3842                 caddr_t saddr, naddr;
3843                 void *tmp = NULL;

3845                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3846                         size_t len;
3847                         size_t npage;
3848                         uint_t prot;
3849                         uintptr_t next;

3851                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3852                         if ((len = naddr - saddr) == 0)
3853                                 continue;
3854                         npage = len / PAGESIZE;
3855                         next = (uintptr_t)(pmp + 1) + round4(npage);
3856                         /*
3857                          * It's possible that the address space can change
3858                          * subtlely even though we're holding as->a_lock
3859                          * due to the nondeterminism of page_exists() in
3860                          * the presence of asychronously flushed pages or
3861                          * mapped files whose sizes are changing.
3862                          * page_exists() may be called indirectly from
3863                          * pr_getprot() by a SEGOP_INCORE() routine.
3864                          * If this happens we need to make sure we don't
3865                          * overrun the buffer whose size we computed based
3866                          * on the initial iteration through the segments.
3867                          * Once we've detected an overflow, we need to clean
3868                          * up the temporary memory allocated in pr_getprot()
3869                          * and retry. If there's a pending signal, we return
3870                          * EINTR so that this thread can be dislodged if
3871                          * a latent bug causes us to spin indefinitely.
3872                          */
3873                         if (next > (uintptr_t)buf + size) {
3874                                 pr_getprot_done(&tmp);
3875                                 AS_LOCK_EXIT(as);
3875                                 AS_LOCK_EXIT(as, &as->a_lock);

3877                                 kmem_free(buf, size);

3879                                 if (ISSIG(curthread, JUSTLOOKING))
3880                                         return (EINTR);

3882                                 goto again;
3883                         }

3885                         php->pr_nmap++;
3886                         php->pr_npage += npage;
3887                         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3888                         pmp->pr_npage = (uint32_t)npage;
3889                         pmp->pr_off = (int32_t)SEGOP_GETOFFSET(seg, saddr);
3890                         pmp->pr_mflags = 0;
3891                         if (prot & PROT_READ)
3892                                 pmp->pr_mflags |= MA_READ;
3893                         if (prot & PROT_WRITE)
3894                                 pmp->pr_mflags |= MA_WRITE;
3895                         if (prot & PROT_EXEC)
3896                                 pmp->pr_mflags |= MA_EXEC;
3897                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3898                                 pmp->pr_mflags |= MA_SHARED;
```



```
3899                         pmp->pr_pagesize = PAGESIZE;
3900                         hat_getstat(as, saddr, len, hatid,
3901                             (char *)(pmp + 1), HAT_SYNC_ZERORM);
3902                         pmp = (ioc_prasmap32_t *)next;
3903                 }
3904                 ASSERT(tmp == NULL);
3905         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3907         AS_LOCK_EXIT(as);
3907         AS_LOCK_EXIT(as, &as->a_lock);

3909         ASSERT((uintptr_t)pmp == (uintptr_t)buf + size);
3910         error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3911         kmem_free(buf, size);

3913         return (error);
3914 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  112225 Wed Nov 25 13:59:36 2015
new/usr/src/uts/common/fs/proc/prsubr.c
patch as-lock-macro-simplification
**********************************************************
_____unchanged_portion_omitted_
1368 #endif  /* _SYSCALL32_IMPL */

1370 /*
1371  * Count the number of segments in this process's address space.
1372  */
1373 int
1374 prnsegs(struct as *as, int reserved)
1375 {
1376         int n = 0;
1377         struct seg *seg;

1379         ASSERT(as != &kas && AS_WRITE_HELD(as));
1379         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1381         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1382                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1383                 caddr_t saddr, naddr;
1384                 void *tmp = NULL;

1386                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1387                         (void) pr_getprot(seg, reserved, &tmp,
1388                             &saddr, &naddr, eaddr);
1389                         if (saddr != naddr)
1390                                 n++;
1391                 }

1393                 ASSERT(tmp == NULL);
1394         }

1396         return (n);
1397 }
_____unchanged_portion_omitted_

1607 /*
1608  * Return an array of structures with memory map information.
1609  * We allocate here; the caller must deallocate.
1610  */
1611 int
1612 prgetmap(proc_t *p, int reserved, list_t *iolhead)
1613 {
1614         struct as *as = p->p_as;
1615         prmap_t *mp;
1616         struct seg *seg;
1617         struct seg *brkseg, *stkseg;
1618         struct vnode *vp;
1619         struct vattr vattr;
1620         uint_t prot;

1622         ASSERT(as != &kas && AS_WRITE_HELD(as));
1622         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1624         /*
1625          * Request an initial buffer size that doesn't waste memory
1626          * if the address space has only a small number of segments.
1627          */
1628         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1630         if ((seg = AS_SEGFIRST(as)) == NULL)
1631                 return (0);
```



```
1633         brkseg = break_seg(p);
1634         stkseg = as_segat(as, prgetstackbase(p));

1636         do {
1637                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1638                 caddr_t saddr, naddr;
1639                 void *tmp = NULL;

1641                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1642                         prot = pr_getprot(seg, reserved, &tmp,
1643                             &saddr, &naddr, eaddr);
1644                         if (saddr == naddr)
1645                                 continue;

1647                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

1649                         mp->pr_vaddr = (uintptr_t)saddr;
1650                         mp->pr_size = naddr - saddr;
1651                         mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1652                         mp->pr_mflags = 0;
1653                         if (prot & PROT_READ)
1654                                 mp->pr_mflags |= MA_READ;
1655                         if (prot & PROT_WRITE)
1656                                 mp->pr_mflags |= MA_WRITE;
1657                         if (prot & PROT_EXEC)
1658                                 mp->pr_mflags |= MA_EXEC;
1659                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1660                                 mp->pr_mflags |= MA_SHARED;
1661                         if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1662                                 mp->pr_mflags |= MA_NORESERVE;
1663                         if (seg->s_ops == &segspt_shmops ||
1664                             (seg->s_ops == &segvn_ops &&
1665                             (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1666                                 mp->pr_mflags |= MA_ANON;
1667                         if (seg == brkseg)
1668                                 mp->pr_mflags |= MA_BREAK;
1669                         else if (seg == stkseg) {
1670                                 mp->pr_mflags |= MA_STACK;
1671                                 if (reserved) {
1672                                         size_t maxstack =
1673                                             ((size_t)p->p_stk_ctl +
1674                                             PAGEOFFSET) & PAGEMASK;
1675                                         mp->pr_vaddr =
1676                                             (uintptr_t)prgetstackbase(p) +
1677                                             p->p_stksize - maxstack;
1678                                         mp->pr_size = (uintptr_t)naddr -
1679                                             mp->pr_vaddr;
1680                                 }
1681                         }
1682                         if (seg->s_ops == &segspt_shmops)
1683                                 mp->pr_mflags |= MA_ISM | MA_SHM;
1684                         mp->pr_pagesize = PAGESIZE;

1686                         /*
1687                          * Manufacture a filename for the "object" directory.
1688                          */
1689                         vattr.va_mask = AT_FSID|AT_NODEID;
1690                         if (seg->s_ops == &segvn_ops &&
1691                             SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1692                             vp != NULL && vp->v_type == VREG &&
1693                             VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1694                                 if (vp == p->p_exec)
1695                                         (void) strcpy(mp->pr_mapname, "a.out");
1696                                 else
1697                                         pr_object_name(mp->pr_mapname,
1698                                             vp, &vattr);
```

```
1699                        }

1701                        /*
1702                         * Get the SysV shared memory id, if any.
1703                         */
1704                        if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1705                            (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1706                            SHMID_NONE) {
1707                                if (mp->pr_shmid == SHMID_FREE)
1708                                        mp->pr_shmid = -1;

1710                                mp->pr_mflags |= MA_SHM;
1711                        } else {
1712                                mp->pr_shmid = -1;
1713                        }
1714                }
1715                ASSERT(tmp == NULL);
1716        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1718        return (0);
1719 }

1721 #ifdef _SYSCALL32_IMPL
1722 int
1723 prgetmap32(proc_t *p, int reserved, list_t *iolhead)
1724 {
1725        struct as *as = p->p_as;
1726        prmap32_t *mp;
1727        struct seg *seg;
1728        struct seg *brkseg, *stkseg;
1729        struct vnode *vp;
1730        struct vattr vattr;
1731        uint_t prot;

1733        ASSERT(as != &kas && AS_WRITE_HELD(as));
1733        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1735        /*
1736         * Request an initial buffer size that doesn't waste memory
1737         * if the address space has only a small number of segments.
1738         */
1739        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1741        if ((seg = AS_SEGFIRST(as)) == NULL)
1742                return (0);

1744        brkseg = break_seg(p);
1745        stkseg = as_segat(as, prgetstackbase(p));

1747        do {
1748                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1749                caddr_t saddr, naddr;
1750                void *tmp = NULL;

1752                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1753                        prot = pr_getprot(seg, reserved, &tmp,
1754                            &saddr, &naddr, eaddr);
1755                        if (saddr == naddr)
1756                                continue;

1758                        mp = pr_iol_newbuf(iolhead, sizeof (*mp));

1760                        mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
1761                        mp->pr_size = (size32_t)(naddr - saddr);
1762                        mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1763                        mp->pr_mflags = 0;
```



```
1764                        if (prot & PROT_READ)
1765                                mp->pr_mflags |= MA_READ;
1766                        if (prot & PROT_WRITE)
1767                                mp->pr_mflags |= MA_WRITE;
1768                        if (prot & PROT_EXEC)
1769                                mp->pr_mflags |= MA_EXEC;
1770                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1771                                mp->pr_mflags |= MA_SHARED;
1772                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1773                                mp->pr_mflags |= MA_NORESERVE;
1774                        if (seg->s_ops == &segspt_shmops ||
1775                            (seg->s_ops == &segvn_ops &&
1776                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1777                                mp->pr_mflags |= MA_ANON;
1778                        if (seg == brkseg)
1779                                mp->pr_mflags |= MA_BREAK;
1780                        else if (seg == stkseg) {
1781                                mp->pr_mflags |= MA_STACK;
1782                                if (reserved) {
1783                                        size_t maxstack =
1784                                            ((size_t)p->p_stk_ctl +
1785                                            PAGEOFFSET) & PAGEMASK;
1786                                        uintptr_t vaddr =
1787                                            (uintptr_t)prgetstackbase(p) +
1788                                            p->p_stksize - maxstack;
1789                                        mp->pr_vaddr = (caddr32_t)vaddr;
1790                                        mp->pr_size = (size32_t)
1791                                            ((uintptr_t)naddr - vaddr);
1792                                }
1793                        }
1794                        if (seg->s_ops == &segspt_shmops)
1795                                mp->pr_mflags |= MA_ISM | MA_SHM;
1796                        mp->pr_pagesize = PAGESIZE;

1798                        /*
1799                         * Manufacture a filename for the "object" directory.
1800                         */
1801                        vattr.va_mask = AT_FSID|AT_NODEID;
1802                        if (seg->s_ops == &segvn_ops &&
1803                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1804                            vp != NULL && vp->v_type == VREG &&
1805                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1806                                if (vp == p->p_exec)
1807                                        (void) strcpy(mp->pr_mapname, "a.out");
1808                                else
1809                                        pr_object_name(mp->pr_mapname,
1810                                            vp, &vattr);
1811                        }

1813                        /*
1814                         * Get the SysV shared memory id, if any.
1815                         */
1816                        if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1817                            (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1818                            SHMID_NONE) {
1819                                if (mp->pr_shmid == SHMID_FREE)
1820                                        mp->pr_shmid = -1;

1822                                mp->pr_mflags |= MA_SHM;
1823                        } else {
1824                                mp->pr_shmid = -1;
1825                        }
1826                }
1827                ASSERT(tmp == NULL);
1828        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
```

```
1830         return (0);
1831 }
1832 #endif  /* _SYSCALL32_IMPL */

1834 /*
1835  * Return the size of the /proc page data file.
1836  */
1837 size_t
1838 prpdsize(struct as *as)
1839 {
1840         struct seg *seg;
1841         size_t size;

1843         ASSERT(as != &kas && AS_WRITE_HELD(as));
1843         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1845         if ((seg = AS_SEGFIRST(as)) == NULL)
1846                 return (0);

1848         size = sizeof (prpageheader_t);
1849         do {
1850                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1851                 caddr_t saddr, naddr;
1852                 void *tmp = NULL;
1853                 size_t npage;

1855                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1856                         (void) pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1857                         if ((npage = (naddr - saddr) / PAGESIZE) != 0)
1858                                 size += sizeof (prasmap_t) + round8(npage);
1859                 }
1860                 ASSERT(tmp == NULL);
1861         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1863         return (size);
1864 }

1866 #ifdef _SYSCALL32_IMPL
1867 size_t
1868 prpdsize32(struct as *as)
1869 {
1870         struct seg *seg;
1871         size_t size;

1873         ASSERT(as != &kas && AS_WRITE_HELD(as));
1873         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1875         if ((seg = AS_SEGFIRST(as)) == NULL)
1876                 return (0);

1878         size = sizeof (prpageheader32_t);
1879         do {
1880                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1881                 caddr_t saddr, naddr;
1882                 void *tmp = NULL;
1883                 size_t npage;

1885                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1886                         (void) pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1887                         if ((npage = (naddr - saddr) / PAGESIZE) != 0)
1888                                 size += sizeof (prasmap32_t) + round8(npage);
1889                 }
1890                 ASSERT(tmp == NULL);
1891         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1893         return (size);
```



```
1894 }
1895 #endif  /* _SYSCALL32_IMPL */

1897 /*
1898  * Read page data information.
1899  */
1900 int
1901 prpdread(proc_t *p, uint_t hatid, struct uio *uiop)
1902 {
1903         struct as *as = p->p_as;
1904         caddr_t buf;
1905         size_t size;
1906         prpageheader_t *php;
1907         prasmap_t *pmp;
1908         struct seg *seg;
1909         int error;

1911 again:
1912         AS_LOCK_ENTER(as, RW_WRITER);
1912         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

1914         if ((seg = AS_SEGFIRST(as)) == NULL) {
1915                 AS_LOCK_EXIT(as);
1915                 AS_LOCK_EXIT(as, &as->a_lock);
1916                 return (0);
1917         }
1918         size = prpdsize(as);
1919         if (uiop->uio_resid < size) {
1920                 AS_LOCK_EXIT(as);
1920                 AS_LOCK_EXIT(as, &as->a_lock);
1921                 return (E2BIG);
1922         }

1924         buf = kmem_zalloc(size, KM_SLEEP);
1925         php = (prpageheader_t *)buf;
1926         pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));

1928         hrt2ts(gethrtime(), &php->pr_tstamp);
1929         php->pr_nmap = 0;
1930         php->pr_npage = 0;
1931         do {
1932                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1933                 caddr_t saddr, naddr;
1934                 void *tmp = NULL;

1936                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1937                         struct vnode *vp;
1938                         struct vattr vattr;
1939                         size_t len;
1940                         size_t npage;
1941                         uint_t prot;
1942                         uintptr_t next;

1944                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1945                         if ((len = (size_t)(naddr - saddr)) == 0)
1946                                 continue;
1947                         npage = len / PAGESIZE;
1948                         next = (uintptr_t)(pmp + 1) + round8(npage);
1949                         /*
1950                          * It's possible that the address space can change
1951                          * subtlely even though we're holding as->a_lock
1952                          * due to the nondeterminism of page_exists() in
1953                          * the presence of asychronously flushed pages or
1954                          * mapped files whose sizes are changing.
1955                          * page_exists() may be called indirectly from
1956                          * pr_getprot() by a SEGOP_INCORE() routine.
```

```
1957                         * If this happens we need to make sure we don't
1958                         * overrun the buffer whose size we computed based
1959                         * on the initial iteration through the segments.
1960                         * Once we've detected an overflow, we need to clean
1961                         * up the temporary memory allocated in pr_getprot()
1962                         * and retry. If there's a pending signal, we return
1963                         * EINTR so that this thread can be dislodged if
1964                         * a latent bug causes us to spin indefinitely.
1965                         */
1966                        if (next > (uintptr_t)buf + size) {
1967                                pr_getprot_done(&tmp);
1968                                AS_LOCK_EXIT(as);
1968                                AS_LOCK_EXIT(as, &as->a_lock);

1970                                kmem_free(buf, size);

1972                                if (ISSIG(curthread, JUSTLOOKING))
1973                                        return (EINTR);

1975                                goto again;
1976                        }

1978                        php->pr_nmap++;
1979                        php->pr_npage += npage;
1980                        pmp->pr_vaddr = (uintptr_t)saddr;
1981                        pmp->pr_npage = npage;
1982                        pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1983                        pmp->pr_mflags = 0;
1984                        if (prot & PROT_READ)
1985                                pmp->pr_mflags |= MA_READ;
1986                        if (prot & PROT_WRITE)
1987                                pmp->pr_mflags |= MA_WRITE;
1988                        if (prot & PROT_EXEC)
1989                                pmp->pr_mflags |= MA_EXEC;
1990                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1991                                pmp->pr_mflags |= MA_SHARED;
1992                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1993                                pmp->pr_mflags |= MA_NORESERVE;
1994                        if (seg->s_ops == &segspt_shmops ||
1995                            (seg->s_ops == &segvn_ops &&
1996                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1997                                pmp->pr_mflags |= MA_ANON;
1998                        if (seg->s_ops == &segspt_shmops)
1999                                pmp->pr_mflags |= MA_ISM | MA_SHM;
2000                        pmp->pr_pagesize = PAGESIZE;
2001                        /*
2002                         * Manufacture a filename for the "object" directory.
2003                         */
2004                        vattr.va_mask = AT_FSID|AT_NODEID;
2005                        if (seg->s_ops == &segvn_ops &&
2006                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2007                            vp != NULL && vp->v_type == VREG &&
2008                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2009                                if (vp == p->p_exec)
2010                                        (void) strcpy(pmp->pr_mapname, "a.out");
2011                                else
2012                                        pr_object_name(pmp->pr_mapname,
2013                                            vp, &vattr);
2014                        }

2016                        /*
2017                         * Get the SysV shared memory id, if any.
2018                         */
2019                        if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2020                            (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2021                            SHMID_NONE) {
```



```
2022                                if (pmp->pr_shmid == SHMID_FREE)
2023                                        pmp->pr_shmid = -1;

2025                                pmp->pr_mflags |= MA_SHM;
2026                        } else {
2027                                pmp->pr_shmid = -1;
2028                        }

2030                        hat_getstat(as, saddr, len, hatid,
2031                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
2032                        pmp = (prasmap_t *)next;
2033                }
2034                ASSERT(tmp == NULL);
2035        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2037        AS_LOCK_EXIT(as);
2037        AS_LOCK_EXIT(as, &as->a_lock);

2039        ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2040        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
2041        kmem_free(buf, size);

2043        return (error);
2044 }

2046 #ifdef _SYSCALL32_IMPL
2047 int
2048 prpdread32(proc_t *p, uint_t hatid, struct uio *uiop)
2049 {
2050        struct as *as = p->p_as;
2051        caddr_t buf;
2052        size_t size;
2053        prpageheader32_t *php;
2054        prasmap32_t *pmp;
2055        struct seg *seg;
2056        int error;

2058 again:
2059        AS_LOCK_ENTER(as, RW_WRITER);
2059        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

2061        if ((seg = AS_SEGFIRST(as)) == NULL) {
2062                AS_LOCK_EXIT(as);
2062                AS_LOCK_EXIT(as, &as->a_lock);
2063                return (0);
2064        }
2065        size = prpdsize32(as);
2066        if (uiop->uio_resid < size) {
2067                AS_LOCK_EXIT(as);
2067                AS_LOCK_EXIT(as, &as->a_lock);
2068                return (E2BIG);
2069        }

2071        buf = kmem_zalloc(size, KM_SLEEP);
2072        php = (prpageheader32_t *)buf;
2073        pmp = (prasmap32_t *)(buf + sizeof (prpageheader32_t));

2075        hrt2ts32(gethrtime(), &php->pr_tstamp);
2076        php->pr_nmap = 0;
2077        php->pr_npage = 0;
2078        do {
2079                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
2080                caddr_t saddr, naddr;
2081                void *tmp = NULL;

2083                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
```

```
2084                        struct vnode *vp;
2085                        struct vattr vattr;
2086                        size_t len;
2087                        size_t npage;
2088                        uint_t prot;
2089                        uintptr_t next;

2091                        prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
2092                        if ((len = (size_t)(naddr - saddr)) == 0)
2093                                continue;
2094                        npage = len / PAGESIZE;
2095                        next = (uintptr_t)(pmp + 1) + round8(npage);
2096                        /*
2097                         * It's possible that the address space can change
2098                         * subtlely even though we're holding as->a_lock
2099                         * due to the nondeterminism of page_exists() in
2100                         * the presence of asychronously flushed pages or
2101                         * mapped files whose sizes are changing.
2102                         * page_exists() may be called indirectly from
2103                         * pr_getprot() by a SEGOP_INCORE() routine.
2104                         * If this happens we need to make sure we don't
2105                         * overrun the buffer whose size we computed based
2106                         * on the initial iteration through the segments.
2107                         * Once we've detected an overflow, we need to clean
2108                         * up the temporary memory allocated in pr_getprot()
2109                         * and retry. If there's a pending signal, we return
2110                         * EINTR so that this thread can be dislodged if
2111                         * a latent bug causes us to spin indefinitely.
2112                         */
2113                        if (next > (uintptr_t)buf + size) {
2114                                pr_getprot_done(&tmp);
2115                                AS_LOCK_EXIT(as);
2115                                AS_LOCK_EXIT(as, &as->a_lock);

2117                                kmem_free(buf, size);

2119                                if (ISSIG(curthread, JUSTLOOKING))
2120                                        return (EINTR);

2122                                goto again;
2123                        }

2125                        php->pr_nmap++;
2126                        php->pr_npage += npage;
2127                        pmp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
2128                        pmp->pr_npage = (size32_t)npage;
2129                        pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
2130                        pmp->pr_mflags = 0;
2131                        if (prot & PROT_READ)
2132                                pmp->pr_mflags |= MA_READ;
2133                        if (prot & PROT_WRITE)
2134                                pmp->pr_mflags |= MA_WRITE;
2135                        if (prot & PROT_EXEC)
2136                                pmp->pr_mflags |= MA_EXEC;
2137                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
2138                                pmp->pr_mflags |= MA_SHARED;
2139                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
2140                                pmp->pr_mflags |= MA_NORESERVE;
2141                        if (seg->s_ops == &segspt_shmops ||
2142                            (seg->s_ops == &segvn_ops &&
2143                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
2144                                pmp->pr_mflags |= MA_ANON;
2145                        if (seg->s_ops == &segspt_shmops)
2146                                pmp->pr_mflags |= MA_ISM | MA_SHM;
2147                        pmp->pr_pagesize = PAGESIZE;
2148                        /*
```



```
2149                         * Manufacture a filename for the "object" directory.
2150                         */
2151                        vattr.va_mask = AT_FSID|AT_NODEID;
2152                        if (seg->s_ops == &segvn_ops &&
2153                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2154                            vp != NULL && vp->v_type == VREG &&
2155                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2156                                if (vp == p->p_exec)
2157                                        (void) strcpy(pmp->pr_mapname, "a.out");
2158                                else
2159                                        pr_object_name(pmp->pr_mapname,
2160                                            vp, &vattr);
2161                        }

2163                        /*
2164                         * Get the SysV shared memory id, if any.
2165                         */
2166                        if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2167                            (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2168                            SHMID_NONE) {
2169                                if (pmp->pr_shmid == SHMID_FREE)
2170                                        pmp->pr_shmid = -1;

2172                                pmp->pr_mflags |= MA_SHM;
2173                        } else {
2174                                pmp->pr_shmid = -1;
2175                        }

2177                        hat_getstat(as, saddr, len, hatid,
2178                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
2179                        pmp = (prasmap32_t *)next;
2180                }
2181                ASSERT(tmp == NULL);
2182        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2184        AS_LOCK_EXIT(as);
2184        AS_LOCK_EXIT(as, &as->a_lock);

2186        ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2187        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
2188        kmem_free(buf, size);

2190        return (error);
2191 }
_____unchanged_portion_omitted_

2215 /*
2216  * Return information used by ps(1).
2217  */
2218 void
2219 prgetpsinfo(proc_t *p, psinfo_t *psp)
2220 {
2221        kthread_t *t;
2222        struct cred *cred;
2223        hrtime_t hrutime, hrstime;

2225        ASSERT(MUTEX_HELD(&p->p_lock));

2227        if ((t = prchoose(p)) == NULL)  /* returns locked thread */
2228                bzero(psp, sizeof (*psp));
2229        else {
2230                thread_unlock(t);
2231                bzero(psp, sizeof (*psp) - sizeof (psp->pr_lwp));
2232        }

2234        /*
```

```
2235          * only export SSYS and SMSACCT; everything else is off-limits to
2236          * userland apps.
2237          */
2238         psp->pr_flag = p->p_flag & (SSYS | SMSACCT);
2239         psp->pr_nlwp = p->p_lwpcnt;
2240         psp->pr_nzomb = p->p_zombcnt;
2241         mutex_enter(&p->p_crlock);
2242         cred = p->p_cred;
2243         psp->pr_uid = crgetruid(cred);
2244         psp->pr_euid = crgetuid(cred);
2245         psp->pr_gid = crgetrgid(cred);
2246         psp->pr_egid = crgetgid(cred);
2247         mutex_exit(&p->p_crlock);
2248         psp->pr_pid = p->p_pid;
2249         if (curproc->p_zone->zone_id != GLOBAL_ZONEID &&
2250             (p->p_flag & SZONETOP)) {
2251                 ASSERT(p->p_zone->zone_id != GLOBAL_ZONEID);
2252                 /*
2253                  * Inside local zones, fake zsched's pid as parent pids for
2254                  * processes which reference processes outside of the zone.
2255                  */
2256                 psp->pr_ppid = curproc->p_zone->zone_zsched->p_pid;
2257         } else {
2258                 psp->pr_ppid = p->p_ppid;
2259         }
2260         psp->pr_pgid = p->p_pgrp;
2261         psp->pr_sid = p->p_sessp->s_sid;
2262         psp->pr_taskid = p->p_task->tk_tkid;
2263         psp->pr_projid = p->p_task->tk_proj->kpj_id;
2264         psp->pr_poolid = p->p_pool->pool_id;
2265         psp->pr_zoneid = p->p_zone->zone_id;
2266         if ((psp->pr_contract = PRCTID(p)) == 0)
2267                 psp->pr_contract = -1;
2268         psp->pr_addr = (uintptr_t)prgetpsaddr(p);
2269         switch (p->p_model) {
2270         case DATAMODEL_ILP32:
2271                 psp->pr_dmodel = PR_MODEL_ILP32;
2272                 break;
2273         case DATAMODEL_LP64:
2274                 psp->pr_dmodel = PR_MODEL_LP64;
2275                 break;
2276         }
2277         hrutime = mstate_aggr_state(p, LMS_USER);
2278         hrstime = mstate_aggr_state(p, LMS_SYSTEM);
2279         hrt2ts((hrutime + hrstime), &psp->pr_time);
2280         TICK_TO_TIMESTRUC(p->p_cutime + p->p_cstime, &psp->pr_ctime);

2282         if (t == NULL) {
2283                 int wcode = p->p_wcode;         /* must be atomic read */

2285                 if (wcode)
2286                         psp->pr_wstat = wstat(wcode, p->p_wdata);
2287                 psp->pr_ttydev = PRNODEV;
2288                 psp->pr_lwp.pr_state = SZOMB;
2289                 psp->pr_lwp.pr_sname = 'Z';
2290                 psp->pr_lwp.pr_bindpro = PBIND_NONE;
2291                 psp->pr_lwp.pr_bindpset = PS_NONE;
2292         } else {
2293                 user_t *up = PTOU(p);
2294                 struct as *as;
2295                 dev_t d;
2296                 extern dev_t rwsconsdev, rconsdev, uconsdev;

2298                 d = cttydev(p);
2299                 /*
2300                  * If the controlling terminal is the real
```



```
2301                  * or workstation console device, map to what the
2302                  * user thinks is the console device. Handle case when
2303                  * rwsconsdev or rconsdev is set to NODEV for Starfire.
2304                  */
2305                 if ((d == rwsconsdev || d == rconsdev) && d != NODEV)
2306                         d = uconsdev;
2307                 psp->pr_ttydev = (d == NODEV) ? PRNODEV : d;
2308                 psp->pr_start = up->u_start;
2309                 bcopy(up->u_comm, psp->pr_fname,
2310                     MIN(sizeof (up->u_comm), sizeof (psp->pr_fname)-1));
2311                 bcopy(up->u_psargs, psp->pr_psargs,
2312                     MIN(PRARGSZ-1, PSARGSZ));
2313                 psp->pr_argc = up->u_argc;
2314                 psp->pr_argv = up->u_argv;
2315                 psp->pr_envp = up->u_envp;

2317                 /* get the chosen lwp's lwpsinfo */
2318                 prgetlwpsinfo(t, &psp->pr_lwp);

2320                 /* compute %cpu for the process */
2321                 if (p->p_lwpcnt == 1)
2322                         psp->pr_pctcpu = psp->pr_lwp.pr_pctcpu;
2323                 else {
2324                         uint64_t pct = 0;
2325                         hrtime_t cur_time = gethrtime_unscaled();

2327                         t = p->p_tlist;
2328                         do {
2329                                 pct += cpu_update_pct(t, cur_time);
2330                         } while ((t = t->t_forw) != p->p_tlist);

2332                         psp->pr_pctcpu = prgetpctcpu(pct);
2333                 }
2334                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
2335                         psp->pr_size = 0;
2336                         psp->pr_rssize = 0;
2337                 } else {
2338                         mutex_exit(&p->p_lock);
2339                         AS_LOCK_ENTER(as, RW_READER);
2339                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2340                         psp->pr_size = btopr(as->a_resvsize) *
2341                             (PAGESIZE / 1024);
2342                         psp->pr_rssize = rm_asrss(as) * (PAGESIZE / 1024);
2343                         psp->pr_pctmem = rm_pctmemory(as);
2344                         AS_LOCK_EXIT(as);
2344                         AS_LOCK_EXIT(as, &as->a_lock);
2345                         mutex_enter(&p->p_lock);
2346                 }
2347         }
2348 }

2350 #ifdef _SYSCALL32_IMPL
2351 void
2352 prgetpsinfo32(proc_t *p, psinfo32_t *psp)
2353 {
2354         kthread_t *t;
2355         struct cred *cred;
2356         hrtime_t hrutime, hrstime;

2358         ASSERT(MUTEX_HELD(&p->p_lock));

2360         if ((t = prchoose(p)) == NULL)  /* returns locked thread */
2361                 bzero(psp, sizeof (*psp));
2362         else {
2363                 thread_unlock(t);
2364                 bzero(psp, sizeof (*psp) - sizeof (psp->pr_lwp));
```

```
2365        }

2367        /*
2368         * only export SSYS and SMSACCT; everything else is off-limits to
2369         * userland apps.
2370         */
2371        psp->pr_flag = p->p_flag & (SSYS | SMSACCT);
2372        psp->pr_nlwp = p->p_lwpcnt;
2373        psp->pr_nzomb = p->p_zombcnt;
2374        mutex_enter(&p->p_crlock);
2375        cred = p->p_cred;
2376        psp->pr_uid = crgetruid(cred);
2377        psp->pr_euid = crgetuid(cred);
2378        psp->pr_gid = crgetrgid(cred);
2379        psp->pr_egid = crgetgid(cred);
2380        mutex_exit(&p->p_crlock);
2381        psp->pr_pid = p->p_pid;
2382        if (curproc->p_zone->zone_id != GLOBAL_ZONEID &&
2383            (p->p_flag & SZONETOP)) {
2384                ASSERT(p->p_zone->zone_id != GLOBAL_ZONEID);
2385                /*
2386                 * Inside local zones, fake zsched's pid as parent pids for
2387                 * processes which reference processes outside of the zone.
2388                 */
2389                psp->pr_ppid = curproc->p_zone->zone_zsched->p_pid;
2390        } else {
2391                psp->pr_ppid = p->p_ppid;
2392        }
2393        psp->pr_pgid = p->p_pgrp;
2394        psp->pr_sid = p->p_sessp->s_sid;
2395        psp->pr_taskid = p->p_task->tk_tkid;
2396        psp->pr_projid = p->p_task->tk_proj->kpj_id;
2397        psp->pr_poolid = p->p_pool->pool_id;
2398        psp->pr_zoneid = p->p_zone->zone_id;
2399        if ((psp->pr_contract = PRCTID(p)) == 0)
2400                psp->pr_contract = -1;
2401        psp->pr_addr = 0;       /* cannot represent 64-bit addr in 32 bits */
2402        switch (p->p_model) {
2403        case DATAMODEL_ILP32:
2404                psp->pr_dmodel = PR_MODEL_ILP32;
2405                break;
2406        case DATAMODEL_LP64:
2407                psp->pr_dmodel = PR_MODEL_LP64;
2408                break;
2409        }
2410        hrutime = mstate_aggr_state(p, LMS_USER);
2411        hrstime = mstate_aggr_state(p, LMS_SYSTEM);
2412        hrt2ts32(hrutime + hrstime, &psp->pr_time);
2413        TICK_TO_TIMESTRUC32(p->p_cutime + p->p_cstime, &psp->pr_ctime);

2415        if (t == NULL) {
2416                extern int wstat(int, int);     /* needs a header file */
2417                int wcode = p->p_wcode;         /* must be atomic read */

2419                if (wcode)
2420                        psp->pr_wstat = wstat(wcode, p->p_wdata);
2421                psp->pr_ttydev = PRNODEV32;
2422                psp->pr_lwp.pr_state = SZOMB;
2423                psp->pr_lwp.pr_sname = 'Z';
2424        } else {
2425                user_t *up = PTOU(p);
2426                struct as *as;
2427                dev_t d;
2428                extern dev_t rwsconsdev, rconsdev, uconsdev;

2430                d = cttydev(p);
```



```
2431                /*
2432                 * If the controlling terminal is the real
2433                 * or workstation console device, map to what the
2434                 * user thinks is the console device. Handle case when
2435                 * rwsconsdev or rconsdev is set to NODEV for Starfire.
2436                 */
2437                if ((d == rwsconsdev || d == rconsdev) && d != NODEV)
2438                        d = uconsdev;
2439                (void) cmpldev(&psp->pr_ttydev, d);
2440                TIMESPEC_TO_TIMESPEC32(&psp->pr_start, &up->u_start);
2441                bcopy(up->u_comm, psp->pr_fname,
2442                    MIN(sizeof (up->u_comm), sizeof (psp->pr_fname)-1));
2443                bcopy(up->u_psargs, psp->pr_psargs,
2444                    MIN(PRARGSZ-1, PSARGSZ));
2445                psp->pr_argc = up->u_argc;
2446                psp->pr_argv = (caddr32_t)up->u_argv;
2447                psp->pr_envp = (caddr32_t)up->u_envp;

2449                /* get the chosen lwp's lwpsinfo */
2450                prgetlwpsinfo32(t, &psp->pr_lwp);

2452                /* compute %cpu for the process */
2453                if (p->p_lwpcnt == 1)
2454                        psp->pr_pctcpu = psp->pr_lwp.pr_pctcpu;
2455                else {
2456                        uint64_t pct = 0;
2457                        hrtime_t cur_time;

2459                        t = p->p_tlist;
2460                        cur_time = gethrtime_unscaled();
2461                        do {
2462                                pct += cpu_update_pct(t, cur_time);
2463                        } while ((t = t->t_forw) != p->p_tlist);

2465                        psp->pr_pctcpu = prgetpctcpu(pct);
2466                }
2467                if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
2468                        psp->pr_size = 0;
2469                        psp->pr_rssize = 0;
2470                } else {
2471                        mutex_exit(&p->p_lock);
2472                        AS_LOCK_ENTER(as, RW_READER);
2472                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2473                        psp->pr_size = (size32_t)
2474                            (btopr(as->a_resvsize) * (PAGESIZE / 1024));
2475                        psp->pr_rssize = (size32_t)
2476                            (rm_asrss(as) * (PAGESIZE / 1024));
2477                        psp->pr_pctmem = rm_pctmemory(as);
2478                        AS_LOCK_EXIT(as);
2478                        AS_LOCK_EXIT(as, &as->a_lock);
2479                        mutex_enter(&p->p_lock);
2480                }
2481        }

2483        /*
2484         * If we are looking at an LP64 process, zero out
2485         * the fields that cannot be represented in ILP32.
2486         */
2487        if (p->p_model != DATAMODEL_ILP32) {
2488                psp->pr_size = 0;
2489                psp->pr_rssize = 0;
2490                psp->pr_argv = 0;
2491                psp->pr_envp = 0;
2492        }
2493 }
______unchanged_portion_omitted_
```

```
3299 /*
3300  * This one is called by the traced process to unwatch all the
3301  * pages while deallocating the list of watched_page structs.
3302  */
3303 void
3304 pr_free_watched_pages(proc_t *p)
3305 {
3306         struct as *as = p->p_as;
3307         struct watched_page *pwp;
3308         uint_t prot;
3309         int    retrycnt, err;
3310         void *cookie;

3312         if (as == NULL || avl_numnodes(&as->a_wpage) == 0)
3313                 return;

3315         ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));
3316         AS_LOCK_ENTER(as, RW_WRITER);
3316         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3318         pwp = avl_first(&as->a_wpage);

3320         cookie = NULL;
3321         while ((pwp = avl_destroy_nodes(&as->a_wpage, &cookie)) != NULL) {
3322                 retrycnt = 0;
3323                 if ((prot = pwp->wp_oprot) != 0) {
3324                         caddr_t addr = pwp->wp_vaddr;
3325                         struct seg *seg;
3326                 retry:

3328                         if ((pwp->wp_prot != prot ||
3329                             (pwp->wp_flags & WP_NOWATCH)) &&
3330                             (seg = as_segat(as, addr)) != NULL) {
3331                                 err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
3332                                 if (err == IE_RETRY) {
3333                                         ASSERT(retrycnt == 0);
3334                                         retrycnt++;
3335                                         goto retry;
3336                                 }
3337                         }
3338                 }
3339                 kmem_free(pwp, sizeof (struct watched_page));
3340         }

3342         avl_destroy(&as->a_wpage);
3343         p->p_wprot = NULL;

3345         AS_LOCK_EXIT(as);
3345         AS_LOCK_EXIT(as, &as->a_lock);
3346 }

3348 /*
3349  * Insert a watched area into the list of watched pages.
3350  * If oflags is zero then we are adding a new watched area.
3351  * Otherwise we are changing the flags of an existing watched area.
3352  */
3353 static int
3354 set_watched_page(proc_t *p, caddr_t vaddr, caddr_t eaddr,
3355         ulong_t flags, ulong_t oflags)
3356 {
3357         struct as *as = p->p_as;
3358         avl_tree_t *pwp_tree;
3359         struct watched_page *pwp, *newpwp;
3360         struct watched_page tpw;
3361         avl_index_t where;
```



```
3362         struct seg *seg;
3363         uint_t prot;
3364         caddr_t addr;

3366         /*
3367          * We need to pre-allocate a list of structures before we grab the
3368          * address space lock to avoid calling kmem_alloc(KM_SLEEP) with locks
3369          * held.
3370          */
3371         newpwp = NULL;
3372         for (addr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3373             addr < eaddr; addr += PAGESIZE) {
3374                 pwp = kmem_zalloc(sizeof (struct watched_page), KM_SLEEP);
3375                 pwp->wp_list = newpwp;
3376                 newpwp = pwp;
3377         }

3379         AS_LOCK_ENTER(as, RW_WRITER);
3379         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3381         /*
3382          * Search for an existing watched page to contain the watched area.
3383          * If none is found, grab a new one from the available list
3384          * and insert it in the active list, keeping the list sorted
3385          * by user-level virtual address.
3386          */
3387         if (p->p_flag & SVFWAIT)
3388                 pwp_tree = &p->p_wpage;
3389         else
3390                 pwp_tree = &as->a_wpage;

3392 again:
3393         if (avl_numnodes(pwp_tree) > prnwatch) {
3394                 AS_LOCK_EXIT(as);
3394                 AS_LOCK_EXIT(as, &as->a_lock);
3395                 while (newpwp != NULL) {
3396                         pwp = newpwp->wp_list;
3397                         kmem_free(newpwp, sizeof (struct watched_page));
3398                         newpwp = pwp;
3399                 }
3400                 return (E2BIG);
3401         }

3403         tpw.wp_vaddr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3404         if ((pwp = avl_find(pwp_tree, &tpw, &where)) == NULL) {
3405                 pwp = newpwp;
3406                 newpwp = newpwp->wp_list;
3407                 pwp->wp_list = NULL;
3408                 pwp->wp_vaddr = (caddr_t)((uintptr_t)vaddr &
3409                     (uintptr_t)PAGEMASK);
3410                 avl_insert(pwp_tree, pwp, where);
3411         }

3413         ASSERT(vaddr >= pwp->wp_vaddr && vaddr < pwp->wp_vaddr + PAGESIZE);

3415         if (oflags & WA_READ)
3416                 pwp->wp_read--;
3417         if (oflags & WA_WRITE)
3418                 pwp->wp_write--;
3419         if (oflags & WA_EXEC)
3420                 pwp->wp_exec--;

3422         ASSERT(pwp->wp_read >= 0);
3423         ASSERT(pwp->wp_write >= 0);
3424         ASSERT(pwp->wp_exec >= 0);
```

```
3426         if (flags & WA_READ)
3427                 pwp->wp_read++;
3428         if (flags & WA_WRITE)
3429                 pwp->wp_write++;
3430         if (flags & WA_EXEC)
3431                 pwp->wp_exec++;

3433         if (!(p->p_flag & SVFWAIT)) {
3434                 vaddr = pwp->wp_vaddr;
3435                 if (pwp->wp_oprot == 0 &&
3436                     (seg = as_segat(as, vaddr)) != NULL) {
3437                         SEGOP_GETPROT(seg, vaddr, 0, &prot);
3438                         pwp->wp_oprot = (uchar_t)prot;
3439                         pwp->wp_prot = (uchar_t)prot;
3440                 }
3441                 if (pwp->wp_oprot != 0) {
3442                         prot = pwp->wp_oprot;
3443                         if (pwp->wp_read)
3444                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3445                         if (pwp->wp_write)
3446                                 prot &= ~PROT_WRITE;
3447                         if (pwp->wp_exec)
3448                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3449                         if (!(pwp->wp_flags & WP_NOWATCH) &&
3450                             pwp->wp_prot != prot &&
3451                             (pwp->wp_flags & WP_SETPROT) == 0) {
3452                                 pwp->wp_flags |= WP_SETPROT;
3453                                 pwp->wp_list = p->p_wprot;
3454                                 p->p_wprot = pwp;
3455                         }
3456                         pwp->wp_prot = (uchar_t)prot;
3457                 }
3458         }

3460         /*
3461          * If the watched area extends into the next page then do
3462          * it over again with the virtual address of the next page.
3463          */
3464         if ((vaddr = pwp->wp_vaddr + PAGESIZE) < eaddr)
3465                 goto again;

3467         AS_LOCK_EXIT(as);
3467         AS_LOCK_EXIT(as, &as->a_lock);

3469         /*
3470          * Free any pages we may have over-allocated
3471          */
3472         while (newpwp != NULL) {
3473                 pwp = newpwp->wp_list;
3474                 kmem_free(newpwp, sizeof (struct watched_page));
3475                 newpwp = pwp;
3476         }

3478         return (0);
3479 }

3481 /*
3482  * Remove a watched area from the list of watched pages.
3483  * A watched area may extend over more than one page.
3484  */
3485 static void
3486 clear_watched_page(proc_t *p, caddr_t vaddr, caddr_t eaddr, ulong_t flags)
3487 {
3488         struct as *as = p->p_as;
3489         struct watched_page *pwp;
3490         struct watched_page tpw;
```



```
3491         avl_tree_t *tree;
3492         avl_index_t where;

3494         AS_LOCK_ENTER(as, RW_WRITER);
3494         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3496         if (p->p_flag & SVFWAIT)
3497                 tree = &p->p_wpage;
3498         else
3499                 tree = &as->a_wpage;

3501         tpw.wp_vaddr = vaddr =
3502             (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3503         pwp = avl_find(tree, &tpw, &where);
3504         if (pwp == NULL)
3505                 pwp = avl_nearest(tree, where, AVL_AFTER);

3507         while (pwp != NULL && pwp->wp_vaddr < eaddr) {
3508                 ASSERT(vaddr <=  pwp->wp_vaddr);

3510                 if (flags & WA_READ)
3511                         pwp->wp_read--;
3512                 if (flags & WA_WRITE)
3513                         pwp->wp_write--;
3514                 if (flags & WA_EXEC)
3515                         pwp->wp_exec--;

3517                 if (pwp->wp_read + pwp->wp_write + pwp->wp_exec != 0) {
3518                         /*
3519                          * Reset the hat layer's protections on this page.
3520                          */
3521                         if (pwp->wp_oprot != 0) {
3522                                 uint_t prot = pwp->wp_oprot;

3524                                 if (pwp->wp_read)
3525                                         prot &=
3526                                             ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3527                                 if (pwp->wp_write)
3528                                         prot &= ~PROT_WRITE;
3529                                 if (pwp->wp_exec)
3530                                         prot &=
3531                                             ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3532                                 if (!(pwp->wp_flags & WP_NOWATCH) &&
3533                                     pwp->wp_prot != prot &&
3534                                     (pwp->wp_flags & WP_SETPROT) == 0) {
3535                                         pwp->wp_flags |= WP_SETPROT;
3536                                         pwp->wp_list = p->p_wprot;
3537                                         p->p_wprot = pwp;
3538                                 }
3539                                 pwp->wp_prot = (uchar_t)prot;
3540                         }
3541                 } else {
3542                         /*
3543                          * No watched areas remain in this page.
3544                          * Reset everything to normal.
3545                          */
3546                         if (pwp->wp_oprot != 0) {
3547                                 pwp->wp_prot = pwp->wp_oprot;
3548                                 if ((pwp->wp_flags & WP_SETPROT) == 0) {
3549                                         pwp->wp_flags |= WP_SETPROT;
3550                                         pwp->wp_list = p->p_wprot;
3551                                         p->p_wprot = pwp;
3552                                 }
3553                         }
3554                 }
```

```
3556                 pwp = AVL_NEXT(tree, pwp);
3557         }

3559         AS_LOCK_EXIT(as);
3559         AS_LOCK_EXIT(as, &as->a_lock);
3560 }

3562 /*
3563  * Return the original protections for the specified page.
3564  */
3565 static void
3566 getwatchprot(struct as *as, caddr_t addr, uint_t *prot)
3567 {
3568         struct watched_page *pwp;
3569         struct watched_page tpw;

3571         ASSERT(AS_LOCK_HELD(as));
3571         ASSERT(AS_LOCK_HELD(as, &as->a_lock));

3573         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3574         if ((pwp = avl_find(&as->a_wpage, &tpw, NULL)) != NULL)
3575                 *prot = pwp->wp_oprot;
3576 }
```

*unchanged_portion_omitted_*

```
3838 uint_t
3839 pr_getprot(struct seg *seg, int reserved, void **tmp,
3840         caddr_t *saddrp, caddr_t *naddrp, caddr_t eaddr)
3841 {
3842         struct as *as = seg->s_as;

3844         caddr_t saddr = *saddrp;
3845         caddr_t naddr;

3847         int check_noreserve;
3848         uint_t prot;

3850         union {
3851                 struct segvn_data *svd;
3852                 struct segdev_data *sdp;
3853                 void *data;
3854         } s;

3856         s.data = seg->s_data;

3858         ASSERT(AS_WRITE_HELD(as));
3858         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
3859         ASSERT(saddr >= seg->s_base && saddr < eaddr);
3860         ASSERT(eaddr <= seg->s_base + seg->s_size);

3862         /*
3863          * Don't include MAP_NORESERVE pages in the address range
3864          * unless their mappings have actually materialized.
3865          * We cheat by knowing that segvn is the only segment
3866          * driver that supports MAP_NORESERVE.
3867          */
3868         check_noreserve =
3869             (!reserved && seg->s_ops == &segvn_ops && s.svd != NULL &&
3870             (s.svd->vp == NULL || s.svd->vp->v_type != VREG) &&
3871             (s.svd->flags & MAP_NORESERVE));

3873         /*
3874          * Examine every page only as a last resort.  We use guilty knowledge
3875          * of segvn and segdev to avoid this: if there are no per-page
3876          * protections present in the segment and we don't care about
3877          * MAP_NORESERVE, then s_data->prot is the prot for the whole segment.
```



```
3878          */
3879         if (!check_noreserve && saddr == seg->s_base &&
3880             seg->s_ops == &segvn_ops && s.svd != NULL && s.svd->pageprot == 0) {
3881                 prot = s.svd->prot;
3882                 getwatchprot(as, saddr, &prot);
3883                 naddr = eaddr;

3885         } else if (saddr == seg->s_base && seg->s_ops == &segdev_ops &&
3886             s.sdp != NULL && s.sdp->pageprot == 0) {
3887                 prot = s.sdp->prot;
3888                 getwatchprot(as, saddr, &prot);
3889                 naddr = eaddr;

3891         } else {
3892                 prpagev_t *pagev;

3894                 /*
3895                  * If addr is sitting at the start of the segment, then
3896                  * create a page vector to store protection and incore
3897                  * information for pages in the segment, and fill it.
3898                  * Otherwise, we expect *tmp to address the prpagev_t
3899                  * allocated by a previous call to this function.
3900                  */
3901                 if (saddr == seg->s_base) {
3902                         pagev = pr_pagev_create(seg, check_noreserve);
3903                         saddr = pr_pagev_fill(pagev, seg, saddr, eaddr);

3905                         ASSERT(*tmp == NULL);
3906                         *tmp = pagev;

3908                         ASSERT(saddr <= eaddr);
3909                         *saddrp = saddr;

3911                         if (saddr == eaddr) {
3912                                 naddr = saddr;
3913                                 prot = 0;
3914                                 goto out;
3915                         }

3917                 } else {
3918                         ASSERT(*tmp != NULL);
3919                         pagev = (prpagev_t *)*tmp;
3920                 }

3922                 naddr = pr_pagev_nextprot(pagev, seg, saddrp, eaddr, &prot);
3923                 ASSERT(naddr <= eaddr);
3924         }

3926 out:
3927         if (naddr == eaddr)
3928                 pr_getprot_done(tmp);
3929         *naddrp = naddr;
3930         return (prot);
3931 }
```

*unchanged_portion_omitted_*

```
3967 static ssize_t
3968 pr_getpagesize(struct seg *seg, caddr_t saddr, caddr_t *naddrp, caddr_t eaddr)
3969 {
3970         ssize_t pagesize, hatsize;

3972         ASSERT(AS_WRITE_HELD(seg->s_as));
3972         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
3973         ASSERT(IS_P2ALIGNED(saddr, PAGESIZE));
3974         ASSERT(IS_P2ALIGNED(eaddr, PAGESIZE));
3975         ASSERT(saddr < eaddr);
```

```
3977         pagesize = hatsize = hat_getpagesize(seg->s_as->a_hat, saddr);
3978         ASSERT(pagesize == -1 || IS_P2ALIGNED(pagesize, pagesize));
3979         ASSERT(pagesize != 0);

3981         if (pagesize == -1)
3982                 pagesize = PAGESIZE;

3984         saddr += P2NPHASE((uintptr_t)saddr, pagesize);

3986         while (saddr < eaddr) {
3987                 if (hatsize != hat_getpagesize(seg->s_as->a_hat, saddr))
3988                         break;
3989                 ASSERT(IS_P2ALIGNED(saddr, pagesize));
3990                 saddr += pagesize;
3991         }

3993         *naddrp = ((saddr < eaddr) ? saddr : eaddr);
3994         return (hatsize);
3995 }

3997 /*
3998  * Return an array of structures with extended memory map information.
3999  * We allocate here; the caller must deallocate.
4000  */
4001 int
4002 prgetxmap(proc_t *p, list_t *iolhead)
4003 {
4004         struct as *as = p->p_as;
4005         prxmap_t *mp;
4006         struct seg *seg;
4007         struct seg *brkseg, *stkseg;
4008         struct vnode *vp;
4009         struct vattr vattr;
4010         uint_t prot;

4012         ASSERT(as != &kas && AS_WRITE_HELD(as));
4012         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4014         /*
4015          * Request an initial buffer size that doesn't waste memory
4016          * if the address space has only a small number of segments.
4017          */
4018         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4020         if ((seg = AS_SEGFIRST(as)) == NULL)
4021                 return (0);

4023         brkseg = break_seg(p);
4024         stkseg = as_segat(as, prgetstackbase(p));

4026         do {
4027                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4028                 caddr_t saddr, naddr, baddr;
4029                 void *tmp = NULL;
4030                 ssize_t psz;
4031                 char *parr;
4032                 uint64_t npages;
4033                 uint64_t pagenum;

4035                 /*
4036                  * Segment loop part one: iterate from the base of the segment
4037                  * to its end, pausing at each address boundary (baddr) between
4038                  * ranges that have different virtual memory protections.
4039                  */
4040                 for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
```



```
4041                         prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4042                         ASSERT(baddr >= saddr && baddr <= eaddr);

4044                         /*
4045                          * Segment loop part two: iterate from the current
4046                          * position to the end of the protection boundary,
4047                          * pausing at each address boundary (naddr) between
4048                          * ranges that have different underlying page sizes.
4049                          */
4050                         for (; saddr < baddr; saddr = naddr) {
4051                                 psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4052                                 ASSERT(naddr >= saddr && naddr <= baddr);

4054                                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4056                                 mp->pr_vaddr = (uintptr_t)saddr;
4057                                 mp->pr_size = naddr - saddr;
4058                                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4059                                 mp->pr_mflags = 0;
4060                                 if (prot & PROT_READ)
4061                                         mp->pr_mflags |= MA_READ;
4062                                 if (prot & PROT_WRITE)
4063                                         mp->pr_mflags |= MA_WRITE;
4064                                 if (prot & PROT_EXEC)
4065                                         mp->pr_mflags |= MA_EXEC;
4066                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4067                                         mp->pr_mflags |= MA_SHARED;
4068                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4069                                         mp->pr_mflags |= MA_NORESERVE;
4070                                 if (seg->s_ops == &segspt_shmops ||
4071                                     (seg->s_ops == &segvn_ops &&
4072                                     (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4073                                     vp == NULL)))
4074                                         mp->pr_mflags |= MA_ANON;
4075                                 if (seg == brkseg)
4076                                         mp->pr_mflags |= MA_BREAK;
4077                                 else if (seg == stkseg)
4078                                         mp->pr_mflags |= MA_STACK;
4079                                 if (seg->s_ops == &segspt_shmops)
4080                                         mp->pr_mflags |= MA_ISM | MA_SHM;

4082                                 mp->pr_pagesize = PAGESIZE;
4083                                 if (psz == -1) {
4084                                         mp->pr_hatpagesize = 0;
4085                                 } else {
4086                                         mp->pr_hatpagesize = psz;
4087                                 }

4089                                 /*
4090                                  * Manufacture a filename for the "object" dir.
4091                                  */
4092                                 mp->pr_dev = PRNODEV;
4093                                 vattr.va_mask = AT_FSID|AT_NODEID;
4094                                 if (seg->s_ops == &segvn_ops &&
4095                                     SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4096                                     vp != NULL && vp->v_type == VREG &&
4097                                     VOP_GETATTR(vp, &vattr, 0, CRED(),
4098                                     NULL) == 0) {
4099                                         mp->pr_dev = vattr.va_fsid;
4100                                         mp->pr_ino = vattr.va_nodeid;
4101                                         if (vp == p->p_exec)
4102                                                 (void) strcpy(mp->pr_mapname,
4103                                                     "a.out");
4104                                         else
4105                                                 pr_object_name(mp->pr_mapname,
4106                                                     vp, &vattr);
```

```
4107                                }

4109                                /*
4110                                 * Get the SysV shared memory id, if any.
4111                                 */
4112                                if ((mp->pr_mflags & MA_SHARED) &&
4113                                    p->p_segacct && (mp->pr_shmid = shmgetid(p,
4114                                    seg->s_base)) != SHMID_NONE) {
4115                                        if (mp->pr_shmid == SHMID_FREE)
4116                                                mp->pr_shmid = -1;

4118                                        mp->pr_mflags |= MA_SHM;
4119                                } else {
4120                                        mp->pr_shmid = -1;
4121                                }

4123                                npages = ((uintptr_t)(naddr - saddr)) >>
4124                                    PAGESHIFT;
4125                                parr = kmem_zalloc(npages, KM_SLEEP);

4127                                SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4129                                for (pagenum = 0; pagenum < npages; pagenum++) {
4130                                        if (parr[pagenum] & SEG_PAGE_INCORE)
4131                                                mp->pr_rss++;
4132                                        if (parr[pagenum] & SEG_PAGE_ANON)
4133                                                mp->pr_anon++;
4134                                        if (parr[pagenum] & SEG_PAGE_LOCKED)
4135                                                mp->pr_locked++;
4136                                }
4137                                kmem_free(parr, npages);
4138                        }
4139                }
4140                ASSERT(tmp == NULL);
4141        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4143        return (0);
4144 }
```

*unchanged_portion_omitted_*

```
4180 #ifdef _SYSCALL32_IMPL
4181 /*
4182  * Return an array of structures with HAT memory map information.
4183  * We allocate here; the caller must deallocate.
4184  */
4185 int
4186 prgetxmap32(proc_t *p, list_t *iolhead)
4187 {
4188        struct as *as = p->p_as;
4189        prxmap32_t *mp;
4190        struct seg *seg;
4191        struct seg *brkseg, *stkseg;
4192        struct vnode *vp;
4193        struct vattr vattr;
4194        uint_t prot;

4196        ASSERT(as != &kas && AS_WRITE_HELD(as));
4196        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4198        /*
4199         * Request an initial buffer size that doesn't waste memory
4200         * if the address space has only a small number of segments.
4201         */
4202        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4204        if ((seg = AS_SEGFIRST(as)) == NULL)
```



```
4205                return (0);

4207        brkseg = break_seg(p);
4208        stkseg = as_segat(as, prgetstackbase(p));

4210        do {
4211                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4212                caddr_t saddr, naddr, baddr;
4213                void *tmp = NULL;
4214                ssize_t psz;
4215                char *parr;
4216                uint64_t npages;
4217                uint64_t pagenum;

4219                /*
4220                 * Segment loop part one: iterate from the base of the segment
4221                 * to its end, pausing at each address boundary (baddr) between
4222                 * ranges that have different virtual memory protections.
4223                 */
4224                for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4225                        prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4226                        ASSERT(baddr >= saddr && baddr <= eaddr);

4228                        /*
4229                         * Segment loop part two: iterate from the current
4230                         * position to the end of the protection boundary,
4231                         * pausing at each address boundary (naddr) between
4232                         * ranges that have different underlying page sizes.
4233                         */
4234                        for (; saddr < baddr; saddr = naddr) {
4235                                psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4236                                ASSERT(naddr >= saddr && naddr <= baddr);

4238                                mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4240                                mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
4241                                mp->pr_size = (size32_t)(naddr - saddr);
4242                                mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4243                                mp->pr_mflags = 0;
4244                                if (prot & PROT_READ)
4245                                        mp->pr_mflags |= MA_READ;
4246                                if (prot & PROT_WRITE)
4247                                        mp->pr_mflags |= MA_WRITE;
4248                                if (prot & PROT_EXEC)
4249                                        mp->pr_mflags |= MA_EXEC;
4250                                if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4251                                        mp->pr_mflags |= MA_SHARED;
4252                                if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4253                                        mp->pr_mflags |= MA_NORESERVE;
4254                                if (seg->s_ops == &segspt_shmops ||
4255                                    (seg->s_ops == &segvn_ops &&
4256                                    (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4257                                    vp == NULL)))
4258                                        mp->pr_mflags |= MA_ANON;
4259                                if (seg == brkseg)
4260                                        mp->pr_mflags |= MA_BREAK;
4261                                else if (seg == stkseg)
4262                                        mp->pr_mflags |= MA_STACK;
4263                                if (seg->s_ops == &segspt_shmops)
4264                                        mp->pr_mflags |= MA_ISM | MA_SHM;

4266                                mp->pr_pagesize = PAGESIZE;
4267                                if (psz == -1) {
4268                                        mp->pr_hatpagesize = 0;
4269                                } else {
4270                                        mp->pr_hatpagesize = psz;
```

```
4271                                }

4273                                /*
4274                                 * Manufacture a filename for the "object" dir.
4275                                 */
4276                                mp->pr_dev = PRNODEV32;
4277                                vattr.va_mask = AT_FSID|AT_NODEID;
4278                                if (seg->s_ops == &segvn_ops &&
4279                                    SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4280                                    vp != NULL && vp->v_type == VREG &&
4281                                    VOP_GETATTR(vp, &vattr, 0, CRED(),
4282                                    NULL) == 0) {
4283                                        (void) cmpldev(&mp->pr_dev,
4284                                            vattr.va_fsid);
4285                                        mp->pr_ino = vattr.va_nodeid;
4286                                        if (vp == p->p_exec)
4287                                                (void) strcpy(mp->pr_mapname,
4288                                                    "a.out");
4289                                        else
4290                                                pr_object_name(mp->pr_mapname,
4291                                                    vp, &vattr);
4292                                }

4294                                /*
4295                                 * Get the SysV shared memory id, if any.
4296                                 */
4297                                if ((mp->pr_mflags & MA_SHARED) &&
4298                                    p->p_segacct && (mp->pr_shmid = shmgetid(p,
4299                                    seg->s_base)) != SHMID_NONE) {
4300                                        if (mp->pr_shmid == SHMID_FREE)
4301                                                mp->pr_shmid = -1;

4303                                        mp->pr_mflags |= MA_SHM;
4304                                } else {
4305                                        mp->pr_shmid = -1;
4306                                }

4308                                npages = ((uintptr_t)(naddr - saddr)) >>
4309                                    PAGESHIFT;
4310                                parr = kmem_zalloc(npages, KM_SLEEP);

4312                                SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4314                                for (pagenum = 0; pagenum < npages; pagenum++) {
4315                                        if (parr[pagenum] & SEG_PAGE_INCORE)
4316                                                mp->pr_rss++;
4317                                        if (parr[pagenum] & SEG_PAGE_ANON)
4318                                                mp->pr_anon++;
4319                                        if (parr[pagenum] & SEG_PAGE_LOCKED)
4320                                                mp->pr_locked++;
4321                                }
4322                                kmem_free(parr, npages);
4323                        }
4324                }
4325                ASSERT(tmp == NULL);
4326        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4328        return (0);
4329 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142610 Wed Nov 25 13:59:36 2015
new/usr/src/uts/common/fs/proc/prvnops.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 872 static int
 873 pr_read_map_common(prnode_t *pnp, uio_t *uiop, prnodetype_t type)
 874 {
 875         proc_t *p;
 876         struct as *as;
 877         list_t iolhead;
 878         int error;

 880 readmap_common:
 881         if ((error = prlock(pnp, ZNO)) != 0)
 882                 return (error);

 884         p = pnp->pr_common->prc_proc;
 885         as = p->p_as;

 887         if ((p->p_flag & SSYS) || as == &kas) {
 888                 prunlock(pnp);
 889                 return (0);
 890         }

 892         if (!AS_LOCK_TRYENTER(as, RW_WRITER)) {
 892         if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_WRITER)) {
 893                 prunlock(pnp);
 894                 delay(1);
 895                 goto readmap_common;
 896         }
 897         mutex_exit(&p->p_lock);

 899         switch (type) {
 900         case PR_XMAP:
 901                 error = prgetxmap(p, &iolhead);
 902                 break;
 903         case PR_RMAP:
 904                 error = prgetmap(p, 1, &iolhead);
 905                 break;
 906         case PR_MAP:
 907                 error = prgetmap(p, 0, &iolhead);
 908                 break;
 909         }

 911         AS_LOCK_EXIT(as);
 911         AS_LOCK_EXIT(as, &as->a_lock);
 912         mutex_enter(&p->p_lock);
 913         prunlock(pnp);

 915         error = pr_iol_uiomove_and_free(&iolhead, uiop, error);

 917         return (error);
 918 }
______unchanged_portion_omitted_

1983 static int
1984 pr_read_map_common_32(prnode_t *pnp, uio_t *uiop, prnodetype_t type)
1985 {
1986         proc_t *p;
1987         struct as *as;
1988         list_t  iolhead;
1989         int error;
```



```
1991 readmap32_common:
1992         if ((error = prlock(pnp, ZNO)) != 0)
1993                 return (error);

1995         p = pnp->pr_common->prc_proc;
1996         as = p->p_as;

1998         if ((p->p_flag & SSYS) || as == &kas) {
1999                 prunlock(pnp);
2000                 return (0);
2001         }

2003         if (PROCESS_NOT_32BIT(p)) {
2004                 prunlock(pnp);
2005                 return (EOVERFLOW);
2006         }

2008         if (!AS_LOCK_TRYENTER(as, RW_WRITER)) {
2008         if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_WRITER)) {
2009                 prunlock(pnp);
2010                 delay(1);
2011                 goto readmap32_common;
2012         }
2013         mutex_exit(&p->p_lock);

2015         switch (type) {
2016         case PR_XMAP:
2017                 error = prgetxmap32(p, &iolhead);
2018                 break;
2019         case PR_RMAP:
2020                 error = prgetmap32(p, 1, &iolhead);
2021                 break;
2022         case PR_MAP:
2023                 error = prgetmap32(p, 0, &iolhead);
2024                 break;
2025         }
2026         AS_LOCK_EXIT(as);
2026         AS_LOCK_EXIT(as, &as->a_lock);
2027         mutex_enter(&p->p_lock);
2028         prunlock(pnp);

2030         error = pr_iol_uiomove_and_free(&iolhead, uiop, error);

2032         return (error);
2033 }
______unchanged_portion_omitted_

2773 static int
2774 prgetattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2775         caller_context_t *ct)
2776 {
2777         prnode_t *pnp = VTOP(vp);
2778         prnodetype_t type = pnp->pr_type;
2779         prcommon_t *pcp;
2780         proc_t *p;
2781         struct as *as;
2782         int error;
2783         vnode_t *rvp;
2784         timestruc_t now;
2785         extern uint_t nproc;
2786         int ngroups;
2787         int nsig;

2789         /*
2790          * This ugly bit of code allows us to keep both versions of this
2791          * function from the same source.
```

```
2792          */
2793 #ifdef _LP64
2794         int iam32bit = (curproc->p_model == DATAMODEL_ILP32);
2795 #define PR_OBJSIZE(obj32, obj64)        \
2796         (iam32bit ? sizeof (obj32) : sizeof (obj64))
2797 #define PR_OBJSPAN(obj32, obj64)        \
2798         (iam32bit ? LSPAN32(obj32) : LSPAN(obj64))
2799 #else
2800 #define PR_OBJSIZE(obj32, obj64)        \
2801         (sizeof (obj64))
2802 #define PR_OBJSPAN(obj32, obj64)        \
2803         (LSPAN(obj64))
2804 #endif

2806         /*
2807          * Return all the attributes.  Should be refined
2808          * so that it returns only those asked for.
2809          * Most of this is complete fakery anyway.
2810          */

2812         /*
2813          * For files in the /proc/<pid>/object directory,
2814          * return the attributes of the underlying object.
2815          * For files in the /proc/<pid>/fd directory,
2816          * return the attributes of the underlying file, but
2817          * make it look inaccessible if it is not a regular file.
2818          * Make directories look like symlinks.
2819          */
2820         switch (type) {
2821         case PR_CURDIR:
2822         case PR_ROOTDIR:
2823                 if (!(flags & ATTR_REAL))
2824                         break;
2825                 /* restrict full knowledge of the attributes to owner or root */
2826                 if ((error = praccess(vp, 0, 0, cr, ct)) != 0)
2827                         return (error);
2828                 /* FALLTHROUGH */
2829         case PR_OBJECT:
2830         case PR_FD:
2831                 rvp = pnp->pr_realvp;
2832                 error = VOP_GETATTR(rvp, vap, flags, cr, ct);
2833                 if (error)
2834                         return (error);
2835                 if (type == PR_FD) {
2836                         if (rvp->v_type != VREG && rvp->v_type != VDIR)
2837                                 vap->va_mode = 0;
2838                         else
2839                                 vap->va_mode &= pnp->pr_mode;
2840                 }
2841                 if (type == PR_OBJECT)
2842                         vap->va_mode &= 07555;
2843                 if (rvp->v_type == VDIR && !(flags & ATTR_REAL)) {
2844                         vap->va_type = VLNK;
2845                         vap->va_size = 0;
2846                         vap->va_nlink = 1;
2847                 }
2848                 return (0);
2849         default:
2850                 break;
2851         }

2853         bzero(vap, sizeof (*vap));
2854         /*
2855          * Large Files: Internally proc now uses VPROC to indicate
2856          * a proc file. Since we have been returning VREG through
2857          * VOP_GETATTR() until now, we continue to do this so as
```



```
2858          * not to break apps depending on this return value.
2859          */
2860         vap->va_type = (vp->v_type == VPROC) ? VREG : vp->v_type;
2861         vap->va_mode = pnp->pr_mode;
2862         vap->va_fsid = vp->v_vfsp->vfs_dev;
2863         vap->va_blksize = DEV_BSIZE;
2864         vap->va_rdev = 0;
2865         vap->va_seq = 0;

2867         if (type == PR_PROCDIR) {
2868                 vap->va_uid = 0;
2869                 vap->va_gid = 0;
2870                 vap->va_nlink = nproc + 2;
2871                 vap->va_nodeid = (ino64_t)PRROOTINO;
2872                 gethrestime(&now);
2873                 vap->va_atime = vap->va_mtime = vap->va_ctime = now;
2874                 vap->va_size = (v.v_proc + 2) * PRSDSIZE;
2875                 vap->va_nblocks = btod(vap->va_size);
2876                 return (0);
2877         }

2879         /*
2880          * /proc/<pid>/self is a symbolic link, and has no prcommon member
2881          */
2882         if (type == PR_SELF) {
2883                 vap->va_uid = crgetruid(CRED());
2884                 vap->va_gid = crgetrgid(CRED());
2885                 vap->va_nodeid = (ino64_t)PR_SELF;
2886                 gethrestime(&now);
2887                 vap->va_atime = vap->va_mtime = vap->va_ctime = now;
2888                 vap->va_nlink = 1;
2889                 vap->va_type = VLNK;
2890                 vap->va_size = 0;
2891                 return (0);
2892         }

2894         p = pr_p_lock(pnp);
2895         mutex_exit(&pr_pidlock);
2896         if (p == NULL)
2897                 return (ENOENT);
2898         pcp = pnp->pr_common;

2900         mutex_enter(&p->p_crlock);
2901         vap->va_uid = crgetruid(p->p_cred);
2902         vap->va_gid = crgetrgid(p->p_cred);
2903         mutex_exit(&p->p_crlock);

2905         vap->va_nlink = 1;
2906         vap->va_nodeid = pnp->pr_ino? pnp->pr_ino :
2907             pmkino(pcp->prc_tslot, pcp->prc_slot, pnp->pr_type);
2908         if ((pcp->prc_flags & PRC_LWP) && pcp->prc_tslot != -1) {
2909                 vap->va_atime.tv_sec = vap->va_mtime.tv_sec =
2910                     vap->va_ctime.tv_sec =
2911                     p->p_lwpdir[pcp->prc_tslot].ld_entry->le_start;
2912                 vap->va_atime.tv_nsec = vap->va_mtime.tv_nsec =
2913                     vap->va_ctime.tv_nsec = 0;
2914         } else {
2915                 user_t *up = PTOU(p);
2916                 vap->va_atime.tv_sec = vap->va_mtime.tv_sec =
2917                     vap->va_ctime.tv_sec = up->u_start.tv_sec;
2918                 vap->va_atime.tv_nsec = vap->va_mtime.tv_nsec =
2919                     vap->va_ctime.tv_nsec = up->u_start.tv_nsec;
2920         }

2922         switch (type) {
2923         case PR_PIDDIR:
```

```
2924                 /* va_nlink: count 'lwp', 'object' and 'fd' directory links */
2925                 vap->va_nlink = 5;
2926                 vap->va_size = sizeof (piddir);
2927                 break;
2928         case PR_OBJECTDIR:
2929                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
2930                         vap->va_size = 2 * PRSDSIZE;
2931                 else {
2932                         mutex_exit(&p->p_lock);
2933                         AS_LOCK_ENTER(as, RW_WRITER);
2933                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2934                         if (as->a_updatedir)
2935                                 rebuild_objdir(as);
2936                         vap->va_size = (as->a_sizedir + 2) * PRSDSIZE;
2937                         AS_LOCK_EXIT(as);
2937                         AS_LOCK_EXIT(as, &as->a_lock);
2938                         mutex_enter(&p->p_lock);
2939                 }
2940                 vap->va_nlink = 2;
2941                 break;
2942         case PR_PATHDIR:
2943                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
2944                         vap->va_size = (P_FINFO(p)->fi_nfiles + 4) * PRSDSIZE;
2945                 else {
2946                         mutex_exit(&p->p_lock);
2947                         AS_LOCK_ENTER(as, RW_WRITER);
2947                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2948                         if (as->a_updatedir)
2949                                 rebuild_objdir(as);
2950                         vap->va_size = (as->a_sizedir + 4 +
2951                             P_FINFO(p)->fi_nfiles) * PRSDSIZE;
2952                         AS_LOCK_EXIT(as);
2952                         AS_LOCK_EXIT(as, &as->a_lock);
2953                         mutex_enter(&p->p_lock);
2954                 }
2955                 vap->va_nlink = 2;
2956                 break;
2957         case PR_PATH:
2958         case PR_CURDIR:
2959         case PR_ROOTDIR:
2960         case PR_CT:
2961                 vap->va_type = VLNK;
2962                 vap->va_size = 0;
2963                 break;
2964         case PR_FDDIR:
2965                 vap->va_nlink = 2;
2966                 vap->va_size = (P_FINFO(p)->fi_nfiles + 2) * PRSDSIZE;
2967                 break;
2968         case PR_LWPDIR:
2969                 /*
2970                  * va_nlink: count each lwp as a directory link.
2971                  * va_size: size of p_lwpdir + 2
2972                  */
2973                 vap->va_nlink = p->p_lwpcnt + p->p_zombcnt + 2;
2974                 vap->va_size = (p->p_lwpdir_sz + 2) * PRSDSIZE;
2975                 break;
2976         case PR_LWPIDDIR:
2977                 vap->va_nlink = 2;
2978                 vap->va_size = sizeof (lwpiddir);
2979                 break;
2980         case PR_CTDIR:
2981                 vap->va_nlink = 2;
2982                 vap->va_size = (avl_numnodes(&p->p_ct_held) + 2) * PRSDSIZE;
2983                 break;
2984         case PR_TMPLDIR:
2985                 vap->va_nlink = 2;
```



```
2986                 vap->va_size = (ct_ntypes + 2) * PRSDSIZE;
2987                 break;
2988         case PR_AS:
2989         case PR_PIDFILE:
2990         case PR_LWPIDFILE:
2991                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
2992                         vap->va_size = 0;
2993                 else
2994                         vap->va_size = as->a_resvsize;
2995                 break;
2996         case PR_STATUS:
2997                 vap->va_size = PR_OBJSIZE(pstatus32_t, pstatus_t);
2998                 break;
2999         case PR_LSTATUS:
3000                 vap->va_size = PR_OBJSIZE(prheader32_t, prheader_t) +
3001                     p->p_lwpcnt * PR_OBJSPAN(lwpstatus32_t, lwpstatus_t);
3002                 break;
3003         case PR_PSINFO:
3004                 vap->va_size = PR_OBJSIZE(psinfo32_t, psinfo_t);
3005                 break;
3006         case PR_LPSINFO:
3007                 vap->va_size = PR_OBJSIZE(prheader32_t, prheader_t) +
3008                     (p->p_lwpcnt + p->p_zombcnt) *
3009                     PR_OBJSPAN(lwpsinfo32_t, lwpsinfo_t);
3010                 break;
3011         case PR_MAP:
3012         case PR_RMAP:
3013         case PR_XMAP:
3014                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
3015                         vap->va_size = 0;
3016                 else {
3017                         mutex_exit(&p->p_lock);
3018                         AS_LOCK_ENTER(as, RW_WRITER);
3018                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3019                         if (type == PR_MAP)
3020                                 vap->va_mtime = as->a_updatetime;
3021                         if (type == PR_XMAP)
3022                                 vap->va_size = prnsegs(as, 0) *
3023                                     PR_OBJSIZE(prxmap32_t, prxmap_t);
3024                         else
3025                                 vap->va_size = prnsegs(as, type == PR_RMAP) *
3026                                     PR_OBJSIZE(prmap32_t, prmap_t);
3027                         AS_LOCK_EXIT(as);
3027                         AS_LOCK_EXIT(as, &as->a_lock);
3028                         mutex_enter(&p->p_lock);
3029                 }
3030                 break;
3031         case PR_CRED:
3032                 mutex_enter(&p->p_crlock);
3033                 vap->va_size = sizeof (prcred_t);
3034                 ngroups = crgetngroups(p->p_cred);
3035                 if (ngroups > 1)
3036                         vap->va_size += (ngroups - 1) * sizeof (gid_t);
3037                 mutex_exit(&p->p_crlock);
3038                 break;
3039         case PR_PRIV:
3040                 vap->va_size = prgetprivsize();
3041                 break;
3042         case PR_SIGACT:
3043                 nsig = PROC_IS_BRANDED(curproc)? BROP(curproc)->b_nsig : NSIG;
3044                 vap->va_size = (nsig-1) *
3045                     PR_OBJSIZE(struct sigaction32, struct sigaction);
3046                 break;
3047         case PR_AUXV:
3048                 vap->va_size = __KERN_NAUXV_IMPL * PR_OBJSIZE(auxv32_t, auxv_t);
3049                 break;
```

```
3050 #if defined(__x86)
3051         case PR_LDT:
3052                 mutex_exit(&p->p_lock);
3053                 mutex_enter(&p->p_ldtlock);
3054                 vap->va_size = prnldt(p) * sizeof (struct ssd);
3055                 mutex_exit(&p->p_ldtlock);
3056                 mutex_enter(&p->p_lock);
3057                 break;
3058 #endif
3059         case PR_USAGE:
3060                 vap->va_size = PR_OBJSIZE(prusage32_t, prusage_t);
3061                 break;
3062         case PR_LUSAGE:
3063                 vap->va_size = PR_OBJSIZE(prheader32_t, prheader_t) +
3064                     (p->p_lwpcnt + 1) * PR_OBJSPAN(prusage32_t, prusage_t);
3065                 break;
3066         case PR_PAGEDATA:
3067                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
3068                         vap->va_size = 0;
3069                 else {
3070                         /*
3071                          * We can drop p->p_lock before grabbing the
3072                          * address space lock because p->p_as will not
3073                          * change while the process is marked P_PR_LOCK.
3074                          */
3075                         mutex_exit(&p->p_lock);
3076                         AS_LOCK_ENTER(as, RW_WRITER);
3076                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3077 #ifdef _LP64
3078                         vap->va_size = iam32bit?
3079                             prpdsize32(as) : prpdsize(as);
3080 #else
3081                         vap->va_size = prpdsize(as);
3082 #endif
3083                         AS_LOCK_EXIT(as);
3083                         AS_LOCK_EXIT(as, &as->a_lock);
3084                         mutex_enter(&p->p_lock);
3085                 }
3086                 break;
3087         case PR_OPAGEDATA:
3088                 if ((p->p_flag & SSYS) || (as = p->p_as) == &kas)
3089                         vap->va_size = 0;
3090                 else {
3091                         mutex_exit(&p->p_lock);
3092                         AS_LOCK_ENTER(as, RW_WRITER);
3092                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3093 #ifdef _LP64
3094                         vap->va_size = iam32bit?
3095                             oprpdsize32(as) : oprpdsize(as);
3096 #else
3097                         vap->va_size = oprpdsize(as);
3098 #endif
3099                         AS_LOCK_EXIT(as);
3099                         AS_LOCK_EXIT(as, &as->a_lock);
3100                         mutex_enter(&p->p_lock);
3101                 }
3102                 break;
3103         case PR_WATCH:
3104                 vap->va_size = avl_numnodes(&p->p_warea) *
3105                     PR_OBJSIZE(prwatch32_t, prwatch_t);
3106                 break;
3107         case PR_LWPSTATUS:
3108                 vap->va_size = PR_OBJSIZE(lwpstatus32_t, lwpstatus_t);
3109                 break;
3110         case PR_LWPSINFO:
3111                 vap->va_size = PR_OBJSIZE(lwpsinfo32_t, lwpsinfo_t);
```



```
3112                 break;
3113         case PR_LWPUSAGE:
3114                 vap->va_size = PR_OBJSIZE(prusage32_t, prusage_t);
3115                 break;
3116         case PR_XREGS:
3117                 if (prhasx(p))
3118                         vap->va_size = prgetprxregsize(p);
3119                 else
3120                         vap->va_size = 0;
3121                 break;
3122         case PR_SPYMASTER:
3123                 if (pnp->pr_common->prc_thread->t_lwp->lwp_spymaster != NULL) {
3124                         vap->va_size = PR_OBJSIZE(psinfo32_t, psinfo_t);
3125                 } else {
3126                         vap->va_size = 0;
3127                 }
3128                 break;
3129 #if defined(__sparc)
3130         case PR_GWINDOWS:
3131         {
3132                 kthread_t *t;
3133                 int n;

3135                 /*
3136                  * If there is no lwp then just make the size zero.
3137                  * This can happen if the lwp exits between the VOP_LOOKUP()
3138                  * of the /proc/<pid>/lwp/<lwpid>/gwindows file and the
3139                  * VOP_GETATTR() of the resulting vnode.
3140                  */
3141                 if ((t = pcp->prc_thread) == NULL) {
3142                         vap->va_size = 0;
3143                         break;
3144                 }
3145                 /*
3146                  * Drop p->p_lock while touching the stack.
3147                  * The P_PR_LOCK flag prevents the lwp from
3148                  * disappearing while we do this.
3149                  */
3150                 mutex_exit(&p->p_lock);
3151                 if ((n = prnwindows(ttolwp(t))) == 0)
3152                         vap->va_size = 0;
3153                 else
3154                         vap->va_size = PR_OBJSIZE(gwindows32_t, gwindows_t) -
3155                             (SPARC_MAXREGWINDOW - n) *
3156                             PR_OBJSIZE(struct rwindow32, struct rwindow);
3157                 mutex_enter(&p->p_lock);
3158                 break;
3159         }
3160         case PR_ASRS:
3161 #ifdef _LP64
3162                 if (p->p_model == DATAMODEL_LP64)
3163                         vap->va_size = sizeof (asrset_t);
3164                 else
3165 #endif
3166                         vap->va_size = 0;
3167                 break;
3168 #endif
3169         case PR_CTL:
3170         case PR_LWPCTL:
3171         default:
3172                 vap->va_size = 0;
3173                 break;
3174         }

3176         prunlock(pnp);
3177         vap->va_nblocks = (fsblkcnt64_t)btod(vap->va_size);
```

```
3178         return (0);
3179 }
_____unchanged_portion_omitted_

3663 static vnode_t *
3664 pr_lookup_objectdir(vnode_t *dp, char *comp)
3665 {
3666         prnode_t *dpnp = VTOP(dp);
3667         prnode_t *pnp;
3668         proc_t *p;
3669         struct seg *seg;
3670         struct as *as;
3671         vnode_t *vp;
3672         vattr_t vattr;

3674         ASSERT(dpnp->pr_type == PR_OBJECTDIR);

3676         pnp = prgetnode(dp, PR_OBJECT);

3678         if (prlock(dpnp, ZNO) != 0) {
3679                 prfreenode(pnp);
3680                 return (NULL);
3681         }
3682         p = dpnp->pr_common->prc_proc;
3683         if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
3684                 prunlock(dpnp);
3685                 prfreenode(pnp);
3686                 return (NULL);
3687         }

3689         /*
3690          * We drop p_lock before grabbing the address space lock
3691          * in order to avoid a deadlock with the clock thread.
3692          * The process will not disappear and its address space
3693          * will not change because it is marked P_PR_LOCK.
3694          */
3695         mutex_exit(&p->p_lock);
3696         AS_LOCK_ENTER(as, RW_READER);
3696         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3697         if ((seg = AS_SEGFIRST(as)) == NULL) {
3698                 vp = NULL;
3699                 goto out;
3700         }
3701         if (strcmp(comp, "a.out") == 0) {
3702                 vp = p->p_exec;
3703                 goto out;
3704         }
3705         do {
3706                 /*
3707                  * Manufacture a filename for the "object" directory.
3708                  */
3709                 vattr.va_mask = AT_FSID|AT_NODEID;
3710                 if (seg->s_ops == &segvn_ops &&
3711                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3712                     vp != NULL && vp->v_type == VREG &&
3713                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
3714                         char name[64];

3716                         if (vp == p->p_exec)    /* "a.out" */
3717                                 continue;
3718                         pr_object_name(name, vp, &vattr);
3719                         if (strcmp(name, comp) == 0)
3720                                 goto out;
3721                 }
3722         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
```



```
3724         vp = NULL;
3725 out:
3726         if (vp != NULL) {
3727                 VN_HOLD(vp);
3728         }
3729         AS_LOCK_EXIT(as);
3729         AS_LOCK_EXIT(as, &as->a_lock);
3730         mutex_enter(&p->p_lock);
3731         prunlock(dpnp);

3733         if (vp == NULL)
3734                 prfreenode(pnp);
3735         else {
3736                 /*
3737                  * Fill in the prnode so future references will
3738                  * be able to find the underlying object's vnode.
3739                  * Don't link this prnode into the list of all
3740                  * prnodes for the process; this is a one-use node.
3741                  * Its use is entirely to catch and fail opens for writing.
3742                  */
3743                 pnp->pr_realvp = vp;
3744                 vp = PTOV(pnp);
3745         }

3747         return (vp);
3748 }
_____unchanged_portion_omitted_

4051 static vnode_t *
4052 pr_lookup_pathdir(vnode_t *dp, char *comp)
4053 {
4054         prnode_t *dpnp = VTOP(dp);
4055         prnode_t *pnp;
4056         vnode_t *vp = NULL;
4057         proc_t *p;
4058         uint_t fd, flags = 0;
4059         int c;
4060         uf_entry_t *ufp;
4061         uf_info_t *fip;
4062         enum { NAME_FD, NAME_OBJECT, NAME_ROOT, NAME_CWD, NAME_UNKNOWN } type;
4063         char *tmp;
4064         int idx;
4065         struct seg *seg;
4066         struct as *as = NULL;
4067         vattr_t vattr;

4069         ASSERT(dpnp->pr_type == PR_PATHDIR);

4071         /*
4072          * First, check if this is a numeric entry, in which case we have a
4073          * file descriptor.
4074          */
4075         fd = 0;
4076         type = NAME_FD;
4077         tmp = comp;
4078         while ((c = *tmp++) != '\0') {
4079                 int ofd;
4080                 if (c < '0' || c > '9') {
4081                         type = NAME_UNKNOWN;
4082                         break;
4083                 }
4084                 ofd = fd;
4085                 fd = 10*fd + c - '0';
4086                 if (fd/10 != ofd) {     /* integer overflow */
4087                         type = NAME_UNKNOWN;
4088                         break;
```

```
4089                    }
4090            }

4092            /*
4093             * Next, see if it is one of the special values {root, cwd}.
4094             */
4095            if (type == NAME_UNKNOWN) {
4096                    if (strcmp(comp, "root") == 0)
4097                            type = NAME_ROOT;
4098                    else if (strcmp(comp, "cwd") == 0)
4099                            type = NAME_CWD;
4100            }

4102            /*
4103             * Grab the necessary data from the process
4104             */
4105            if (prlock(dpnp, ZNO) != 0)
4106                    return (NULL);
4107            p = dpnp->pr_common->prc_proc;

4109            fip = P_FINFO(p);

4111            switch (type) {
4112            case NAME_ROOT:
4113                    if ((vp = PTOU(p)->u_rdir) == NULL)
4114                            vp = p->p_zone->zone_rootvp;
4115                    VN_HOLD(vp);
4116                    break;
4117            case NAME_CWD:
4118                    vp = PTOU(p)->u_cdir;
4119                    VN_HOLD(vp);
4120                    break;
4121            default:
4122                    if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
4123                            prunlock(dpnp);
4124                            return (NULL);
4125                    }
4126            }
4127            mutex_exit(&p->p_lock);

4129            /*
4130             * Determine if this is an object entry
4131             */
4132            if (type == NAME_UNKNOWN) {
4133                    /*
4134                     * Start with the inode index immediately after the number of
4135                     * files.
4136                     */
4137                    mutex_enter(&fip->fi_lock);
4138                    idx = fip->fi_nfiles + 4;
4139                    mutex_exit(&fip->fi_lock);

4141                    if (strcmp(comp, "a.out") == 0) {
4142                            if (p->p_execdir != NULL) {
4143                                    vp = p->p_execdir;
4144                                    VN_HOLD(vp);
4145                                    type = NAME_OBJECT;
4146                                    flags |= PR_AOUT;
4147                            } else {
4148                                    vp = p->p_exec;
4149                                    VN_HOLD(vp);
4150                                    type = NAME_OBJECT;
4151                            }
4152                    } else {
4153                            AS_LOCK_ENTER(as, RW_READER);
4153                            AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
```



```
4154                            if ((seg = AS_SEGFIRST(as)) != NULL) {
4155                                    do {
4156                                            /*
4157                                             * Manufacture a filename for the
4158                                             * "object" directory.
4159                                             */
4160                                            vattr.va_mask = AT_FSID|AT_NODEID;
4161                                            if (seg->s_ops == &segvn_ops &&
4162                                                SEGOP_GETVP(seg, seg->s_base, &vp)
4163                                                == 0 &&
4164                                                vp != NULL && vp->v_type == VREG &&
4165                                                VOP_GETATTR(vp, &vattr, 0, CRED(),
4166                                                NULL) == 0) {
4167                                                    char name[64];

4169                                                    if (vp == p->p_exec)
4170                                                            continue;
4171                                                    idx++;
4172                                                    pr_object_name(name, vp,
4173                                                        &vattr);
4174                                                    if (strcmp(name, comp) == 0)
4175                                                            break;
4176                                            }
4177                                    } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4178                            }

4180                            if (seg == NULL) {
4181                                    vp = NULL;
4182                            } else {
4183                                    VN_HOLD(vp);
4184                                    type = NAME_OBJECT;
4185                            }

4187                            AS_LOCK_EXIT(as);
4187                            AS_LOCK_EXIT(as, &as->a_lock);
4188                    }
4189            }


4192            switch (type) {
4193            case NAME_FD:
4194                    mutex_enter(&fip->fi_lock);
4195                    if (fd < fip->fi_nfiles) {
4196                            UF_ENTER(ufp, fip, fd);
4197                            if (ufp->uf_file != NULL) {
4198                                    vp = ufp->uf_file->f_vnode;
4199                                    VN_HOLD(vp);
4200                            }
4201                            UF_EXIT(ufp);
4202                    }
4203                    mutex_exit(&fip->fi_lock);
4204                    idx = fd + 4;
4205                    break;
4206            case NAME_ROOT:
4207                    idx = 2;
4208                    break;
4209            case NAME_CWD:
4210                    idx = 3;
4211                    break;
4212            case NAME_OBJECT:
4213            case NAME_UNKNOWN:
4214                    /* Nothing to do */
4215                    break;
4216            }

4218            mutex_enter(&p->p_lock);
```

```
4219         prunlock(dpnp);
4220
4221         if (vp != NULL) {
4222                 pnp = prgetnode(dp, PR_PATH);
4223
4224                 pnp->pr_flags |= flags;
4225                 pnp->pr_common = dpnp->pr_common;
4226                 pnp->pr_pcommon = dpnp->pr_pcommon;
4227                 pnp->pr_realvp = vp;
4228                 pnp->pr_parent = dp;            /* needed for prlookup */
4229                 pnp->pr_ino = pmkino(idx, dpnp->pr_common->prc_slot, PR_PATH);
4230                 VN_HOLD(dp);
4231                 vp = PTOV(pnp);
4232                 vp->v_type = VLNK;
4233         }
4234
4235         return (vp);
4236 }
______unchanged_portion_omitted_

4829 static void
4830 rebuild_objdir(struct as *as)
4831 {
4832         struct seg *seg;
4833         vnode_t *vp;
4834         vattr_t vattr;
4835         vnode_t **dir;
4836         ulong_t nalloc;
4837         ulong_t nentries;
4838         int i, j;
4839         ulong_t nold, nnew;
4840
4841         ASSERT(AS_WRITE_HELD(as));
4841         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
4842
4843         if (as->a_updatedir == 0 && as->a_objectdir != NULL)
4844                 return;
4845         as->a_updatedir = 0;
4846
4847         if ((nalloc = avl_numnodes(&as->a_segtree)) == 0 ||
4848             (seg = AS_SEGFIRST(as)) == NULL)    /* can't happen? */
4849                 return;
4850
4851         /*
4852          * Allocate space for the new object directory.
4853          * (This is usually about two times too many entries.)
4854          */
4855         nalloc = (nalloc + 0xf) & ~0xf;         /* multiple of 16 */
4856         dir = kmem_zalloc(nalloc * sizeof (vnode_t *), KM_SLEEP);
4857
4858         /* fill in the new directory with desired entries */
4859         nentries = 0;
4860         do {
4861                 vattr.va_mask = AT_FSID|AT_NODEID;
4862                 if (seg->s_ops == &segvn_ops &&
4863                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
4864                     vp != NULL && vp->v_type == VREG &&
4865                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4866                         for (i = 0; i < nentries; i++)
4867                                 if (vp == dir[i])
4868                                         break;
4869                         if (i == nentries) {
4870                                 ASSERT(nentries < nalloc);
4871                                 dir[nentries++] = vp;
4872                         }
4873                 }
```



```
4874         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4875
4876         if (as->a_objectdir == NULL) {  /* first time */
4877                 as->a_objectdir = dir;
4878                 as->a_sizedir = nalloc;
4879                 return;
4880         }
4881
4882         /*
4883          * Null out all of the defunct entries in the old directory.
4884          */
4885         nold = 0;
4886         nnew = nentries;
4887         for (i = 0; i < as->a_sizedir; i++) {
4888                 if ((vp = as->a_objectdir[i]) != NULL) {
4889                         for (j = 0; j < nentries; j++) {
4890                                 if (vp == dir[j]) {
4891                                         dir[j] = NULL;
4892                                         nnew--;
4893                                         break;
4894                                 }
4895                         }
4896                         if (j == nentries)
4897                                 as->a_objectdir[i] = NULL;
4898                         else
4899                                 nold++;
4900                 }
4901         }
4902
4903         if (nold + nnew > as->a_sizedir) {
4904                 /*
4905                  * Reallocate the old directory to have enough
4906                  * space for the old and new entries combined.
4907                  * Round up to the next multiple of 16.
4908                  */
4909                 ulong_t newsize = (nold + nnew + 0xf) & ~0xf;
4910                 vnode_t **newdir = kmem_zalloc(newsize * sizeof (vnode_t *),
4911                     KM_SLEEP);
4912                 bcopy(as->a_objectdir, newdir,
4913                     as->a_sizedir * sizeof (vnode_t *));
4914                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
4915                 as->a_objectdir = newdir;
4916                 as->a_sizedir = newsize;
4917         }
4918
4919         /*
4920          * Move all new entries to the old directory and
4921          * deallocate the space used by the new directory.
4922          */
4923         if (nnew) {
4924                 for (i = 0, j = 0; i < nentries; i++) {
4925                         if ((vp = dir[i]) == NULL)
4926                                 continue;
4927                         for (; j < as->a_sizedir; j++) {
4928                                 if (as->a_objectdir[j] != NULL)
4929                                         continue;
4930                                 as->a_objectdir[j++] = vp;
4931                                 break;
4932                         }
4933                 }
4934         }
4935         kmem_free(dir, nalloc * sizeof (vnode_t *));
4936 }
4937
4938 /*
4939  * Return the vnode from a slot in the process's object directory.
```

```
4940  * The caller must have locked the process's address space.
4941  * The only caller is below, in pr_readdir_objectdir().
4942  */
4943 static vnode_t *
4944 obj_entry(struct as *as, int slot)
4945 {
4946         ASSERT(AS_LOCK_HELD(as));
4946         ASSERT(AS_LOCK_HELD(as, &as->a_lock));
4947         if (as->a_objectdir == NULL)
4948                 return (NULL);
4949         ASSERT(slot < as->a_sizedir);
4950         return (as->a_objectdir[slot]);
4951 }

4953 /* ARGSUSED */
4954 static int
4955 pr_readdir_objectdir(prnode_t *pnp, uio_t *uiop, int *eofp)
4956 {
4957         gfs_readdir_state_t gstate;
4958         int error, eof = 0;
4959         offset_t n;
4960         int pslot;
4961         size_t objdirsize;
4962         proc_t *p;
4963         struct as *as;
4964         vnode_t *vp;

4966         ASSERT(pnp->pr_type == PR_OBJECTDIR);

4968         if ((error = prlock(pnp, ZNO)) != 0)
4969                 return (error);
4970         p = pnp->pr_common->prc_proc;
4971         pslot = p->p_slot;

4973         /*
4974          * We drop p_lock before grabbing the address space lock
4975          * in order to avoid a deadlock with the clock thread.
4976          * The process will not disappear and its address space
4977          * will not change because it is marked P_PR_LOCK.
4978          */
4979         mutex_exit(&p->p_lock);

4981         if ((error = gfs_readdir_init(&gstate, 64, PRSDSIZE, uiop,
4982             pmkino(0, pslot, PR_PIDDIR),
4983             pmkino(0, pslot, PR_OBJECTDIR), 0)) != 0) {
4984                 mutex_enter(&p->p_lock);
4985                 prunlock(pnp);
4986                 return (error);
4987         }

4989         if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
4990                 as = NULL;
4991                 objdirsize = 0;
4992         }

4994         /*
4995          * Loop until user's request is satisfied or until
4996          * all mapped objects have been examined. Cannot hold
4997          * the address space lock for the following call as
4998          * gfs_readdir_pred() utimately causes a call to uiomove().
4999          */
5000         while ((error = gfs_readdir_pred(&gstate, uiop, &n)) == 0) {
5001                 vattr_t vattr;
5002                 char str[64];

5004                 /*
```



```
5005                  * Set the correct size of the directory just
5006                  * in case the process has changed it's address
5007                  * space via mmap/munmap calls.
5008                  */
5009                 if (as != NULL) {
5010                         AS_LOCK_ENTER(as, RW_WRITER);
5010                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
5011                         if (as->a_updatedir)
5012                                 rebuild_objdir(as);
5013                         objdirsize = as->a_sizedir;
5014                 }

5016                 /*
5017                  * Find next object.
5018                  */
5019                 vattr.va_mask = AT_FSID | AT_NODEID;
5020                 while (n < objdirsize && (((vp = obj_entry(as, n)) == NULL) ||
5021                     (VOP_GETATTR(vp, &vattr, 0, CRED(), NULL)
5022                     != 0))) {
5023                         vattr.va_mask = AT_FSID | AT_NODEID;
5024                         n++;
5025                 }

5027                 if (as != NULL)
5028                         AS_LOCK_EXIT(as);
5028                         AS_LOCK_EXIT(as, &as->a_lock);

5030                 /*
5031                  * Stop when all objects have been reported.
5032                  */
5033                 if (n >= objdirsize) {
5034                         eof = 1;
5035                         break;
5036                 }

5038                 if (vp == p->p_exec)
5039                         (void) strcpy(str, "a.out");
5040                 else
5041                         pr_object_name(str, vp, &vattr);

5043                 error = gfs_readdir_emit(&gstate, uiop, n, vattr.va_nodeid,
5044                     str, 0);

5046                 if (error)
5047                         break;
5048         }

5050         mutex_enter(&p->p_lock);
5051         prunlock(pnp);

5053         return (gfs_readdir_fini(&gstate, error, eofp, eof));
5054 }
______unchanged_portion_omitted_

5262 /* ARGSUSED */
5263 static int
5264 pr_readdir_pathdir(prnode_t *pnp, uio_t *uiop, int *eofp)
5265 {
5266         longlong_t bp[DIRENT64_RECLEN(64) / sizeof (longlong_t)];
5267         dirent64_t *dirent = (dirent64_t *)bp;
5268         int reclen;
5269         ssize_t oresid;
5270         offset_t off, idx;
5271         int error = 0;
5272         proc_t *p;
5273         int fd, obj;
```

```
5274        int pslot;
5275        int fddirsize;
5276        uf_info_t *fip;
5277        struct as *as = NULL;
5278        size_t objdirsize;
5279        vattr_t vattr;
5280        vnode_t *vp;

5282        ASSERT(pnp->pr_type == PR_PATHDIR);

5284        if (uiop->uio_offset < 0 ||
5285            uiop->uio_resid <= 0 ||
5286            (uiop->uio_offset % PRSDSIZE) != 0)
5287                return (EINVAL);
5288        oresid = uiop->uio_resid;
5289        bzero(bp, sizeof (bp));

5291        if ((error = prlock(pnp, ZNO)) != 0)
5292                return (error);
5293        p = pnp->pr_common->prc_proc;
5294        fip = P_FINFO(p);
5295        pslot = p->p_slot;
5296        mutex_exit(&p->p_lock);

5298        if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
5299                as = NULL;
5300                objdirsize = 0;
5301        } else {
5302                AS_LOCK_ENTER(as, RW_WRITER);
5302                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
5303                if (as->a_updatedir)
5304                        rebuild_objdir(as);
5305                objdirsize = as->a_sizedir;
5306                AS_LOCK_EXIT(as);
5306                AS_LOCK_EXIT(as, &as->a_lock);
5307                as = NULL;
5308        }

5310        mutex_enter(&fip->fi_lock);
5311        if ((p->p_flag & SSYS) || p->p_as == &kas)
5312                fddirsize = 0;
5313        else
5314                fddirsize = fip->fi_nfiles;

5316        for (; uiop->uio_resid > 0; uiop->uio_offset = off + PRSDSIZE) {
5317                /*
5318                 * There are 4 special files in the path directory: ".", "..",
5319                 * "root", and "cwd".  We handle those specially here.
5320                 */
5321                off = uiop->uio_offset;
5322                idx = off / PRSDSIZE;
5323                if (off == 0) {                         /* "." */
5324                        dirent->d_ino = pmkino(0, pslot, PR_PATHDIR);
5325                        dirent->d_name[0] = '.';
5326                        dirent->d_name[1] = '\0';
5327                        reclen = DIRENT64_RECLEN(1);
5328                } else if (idx == 1) {                  /* ".." */
5329                        dirent->d_ino = pmkino(0, pslot, PR_PIDDIR);
5330                        dirent->d_name[0] = '.';
5331                        dirent->d_name[1] = '.';
5332                        dirent->d_name[2] = '\0';
5333                        reclen = DIRENT64_RECLEN(2);
5334                } else if (idx == 2) {                  /* "root" */
5335                        dirent->d_ino = pmkino(idx, pslot, PR_PATH);
5336                        (void) strcpy(dirent->d_name, "root");
5337                        reclen = DIRENT64_RECLEN(4);
```



```
5338                } else if (idx == 3) {                  /* "cwd" */
5339                        dirent->d_ino = pmkino(idx, pslot, PR_PATH);
5340                        (void) strcpy(dirent->d_name, "cwd");
5341                        reclen = DIRENT64_RECLEN(3);
5342                } else if (idx < 4 + fddirsize) {
5343                        /*
5344                         * In this case, we have one of the file descriptors.
5345                         */
5346                        fd = idx - 4;
5347                        if (fip->fi_list[fd].uf_file == NULL)
5348                                continue;
5349                        dirent->d_ino = pmkino(idx, pslot, PR_PATH);
5350                        (void) pr_u32tos(fd, dirent->d_name, PLNSIZ+1);
5351                        reclen = DIRENT64_RECLEN(PLNSIZ);
5352                } else if (idx < 4 + fddirsize + objdirsize) {
5353                        if (fip != NULL) {
5354                                mutex_exit(&fip->fi_lock);
5355                                fip = NULL;
5356                        }

5358                        /*
5359                         * We drop p_lock before grabbing the address space lock
5360                         * in order to avoid a deadlock with the clock thread.
5361                         * The process will not disappear and its address space
5362                         * will not change because it is marked P_PR_LOCK.
5363                         */
5364                        if (as == NULL) {
5365                                as = p->p_as;
5366                                AS_LOCK_ENTER(as, RW_WRITER);
5366                                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
5367                        }

5369                        if (as->a_updatedir) {
5370                                rebuild_objdir(as);
5371                                objdirsize = as->a_sizedir;
5372                        }

5374                        obj = idx - 4 - fddirsize;
5375                        if ((vp = obj_entry(as, obj)) == NULL)
5376                                continue;
5377                        vattr.va_mask = AT_FSID|AT_NODEID;
5378                        if (VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) != 0)
5379                                continue;
5380                        if (vp == p->p_exec)
5381                                (void) strcpy(dirent->d_name, "a.out");
5382                        else
5383                                pr_object_name(dirent->d_name, vp, &vattr);
5384                        dirent->d_ino = pmkino(idx, pslot, PR_PATH);
5385                        reclen = DIRENT64_RECLEN(strlen(dirent->d_name));
5386                } else {
5387                        break;
5388                }

5390                dirent->d_off = uiop->uio_offset + PRSDSIZE;
5391                dirent->d_reclen = (ushort_t)reclen;
5392                if (reclen > uiop->uio_resid) {
5393                        /*
5394                         * Error if no entries have been returned yet.
5395                         */
5396                        if (uiop->uio_resid == oresid)
5397                                error = EINVAL;
5398                        break;
5399                }
5400                /*
5401                 * Drop the address space lock to do the uiomove().
5402                 */
```

```
5403                 if (as != NULL)
5404                         AS_LOCK_EXIT(as);
5404                         AS_LOCK_EXIT(as, &as->a_lock);

5406                 error = uiomove((caddr_t)dirent, reclen, UIO_READ, uiop);
5407                 if (as != NULL)
5408                         AS_LOCK_ENTER(as, RW_WRITER);
5408                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

5410                 if (error)
5411                         break;
5412         }

5414         if (error == 0 && eofp)
5415                 *eofp = (uiop->uio_offset >= (fddirsize + 2) * PRSDSIZE);

5417         if (fip != NULL)
5418                 mutex_exit(&fip->fi_lock);
5419         if (as != NULL)
5420                 AS_LOCK_EXIT(as);
5420                 AS_LOCK_EXIT(as, &as->a_lock);
5421         mutex_enter(&p->p_lock);
5422         prunlock(pnp);
5423         return (error);
5424 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  170793 Wed Nov 25 13:59:36 2015
new/usr/src/uts/common/fs/ufs/ufs_vnops.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

5587 uint64_t ufs_map_alock_retry_cnt;
5588 uint64_t ufs_map_lockfs_retry_cnt;

5590 /* ARGSUSED */
5591 static int
5592 ufs_map(struct vnode *vp,
5593         offset_t off,
5594         struct as *as,
5595         caddr_t *addrp,
5596         size_t len,
5597         uchar_t prot,
5598         uchar_t maxprot,
5599         uint_t flags,
5600         struct cred *cr,
5601         caller_context_t *ct)
5602 {
5603         struct segvn_crargs vn_a;
5604         struct ufsvfs *ufsvfsp = VTOI(vp)->i_ufsvfs;
5605         struct ulockfs *ulp;
5606         int error, sig;
5607         k_sigset_t smask;
5608         caddr_t hint = *addrp;

5610         if (vp->v_flag & VNOMAP) {
5611                 error = ENOSYS;
5612                 goto out;
5613         }

5615         if (off < (offset_t)0 || (offset_t)(off + len) < (offset_t)0) {
5616                 error = ENXIO;
5617                 goto out;
5618         }

5620         if (vp->v_type != VREG) {
5621                 error = ENODEV;
5622                 goto out;
5623         }

5625 retry_map:
5626         *addrp = hint;
5627         /*
5628          * If file is being locked, disallow mapping.
5629          */
5630         if (vn_has_mandatory_locks(vp, VTOI(vp)->i_mode)) {
5631                 error = EAGAIN;
5632                 goto out;
5633         }

5635         as_rangelock(as);
5636         /*
5637          * Note that if we are retrying (because ufs_lockfs_trybegin failed in
5638          * the previous attempt), some other thread could have grabbed
5639          * the same VA range if MAP_FIXED is set. In that case, choose_addr
5640          * would unmap the valid VA range, that is ok.
5641          */
5642         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
5643         if (error != 0) {
5644                 as_rangeunlock(as);
5645                 goto out;
```



```
5646         }

5648         /*
5649          * a_lock has to be acquired before entering the lockfs protocol
5650          * because that is the order in which pagefault works. Also we cannot
5651          * block on a_lock here because this waiting writer will prevent
5652          * further readers like ufs_read from progressing and could cause
5653          * deadlock between ufs_read/ufs_map/pagefault when a quiesce is
5654          * pending.
5655          */
5656         while (!AS_LOCK_TRYENTER(as, RW_WRITER)) {
5656         while (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_WRITER)) {
5657                 ufs_map_alock_retry_cnt++;
5658                 delay(RETRY_LOCK_DELAY);
5659         }

5661         /*
5662          * We can't hold as->a_lock and wait for lockfs to succeed because
5663          * the proc tools might hang on a_lock, so call ufs_lockfs_trybegin()
5664          * instead.
5665          */
5666         if (error = ufs_lockfs_trybegin(ufsvfsp, &ulp, ULOCKFS_MAP_MASK)) {
5667                 /*
5668                  * ufs_lockfs_trybegin() did not succeed. It is safer to give up
5669                  * as->a_lock and wait for ulp->ul_fs_lock status to change.
5670                  */
5671                 ufs_map_lockfs_retry_cnt++;
5672                 AS_LOCK_EXIT(as);
5672                 AS_LOCK_EXIT(as, &as->a_lock);
5673                 as_rangeunlock(as);
5674                 if (error == EIO)
5675                         goto out;

5677                 mutex_enter(&ulp->ul_lock);
5678                 while (ulp->ul_fs_lock & ULOCKFS_MAP_MASK) {
5679                         if (ULOCKFS_IS_SLOCK(ulp) || ufsvfsp->vfs_nointr) {
5680                                 cv_wait(&ulp->ul_cv, &ulp->ul_lock);
5681                         } else {
5682                                 sigintr(&smask, 1);
5683                                 sig = cv_wait_sig(&ulp->ul_cv, &ulp->ul_lock);
5684                                 sigunintr(&smask);
5685                                 if (((ulp->ul_fs_lock & ULOCKFS_MAP_MASK) &&
5686                                     !sig) || ufsvfsp->vfs_dontblock) {
5687                                         mutex_exit(&ulp->ul_lock);
5688                                         return (EINTR);
5689                                 }
5690                         }
5691                 }
5692                 mutex_exit(&ulp->ul_lock);
5693                 goto retry_map;
5694         }

5696         vn_a.vp = vp;
5697         vn_a.offset = (u_offset_t)off;
5698         vn_a.type = flags & MAP_TYPE;
5699         vn_a.prot = prot;
5700         vn_a.maxprot = maxprot;
5701         vn_a.cred = cr;
5702         vn_a.amp = NULL;
5703         vn_a.flags = flags & ~MAP_TYPE;
5704         vn_a.szc = 0;
5705         vn_a.lgrp_mem_policy_flags = 0;

5707         error = as_map_locked(as, *addrp, len, segvn_create, &vn_a);
5708         if (ulp)
5709                 ufs_lockfs_end(ulp);
```

```
5710         as_rangeunlock(as);
5711 out:
5712         return (error);
5713 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   23616 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/io/mem.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 285 static int
 286 mmpagelock(struct as *as, caddr_t va)
 287 {
 288         struct seg *seg;
 289         int i;

 291         AS_LOCK_ENTER(as, RW_READER);
 291         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 292         seg = as_segat(as, va);
 293         i = (seg != NULL)? SEGOP_CAPABLE(seg, S_CAPABILITY_NOMINFLT) : 0;
 294         AS_LOCK_EXIT(as);
 294         AS_LOCK_EXIT(as, &as->a_lock);

 296         return (i);
 297 }
______unchanged_portion_omitted_

 436 /*
 437  * Private ioctl for libkvm to support kvm_physaddr().
 438  * Given an address space and a VA, compute the PA.
 439  */
 440 static int
 441 mmioctl_vtop(intptr_t data)
 442 {
 443 #ifdef _SYSCALL32
 444         mem_vtop32_t vtop32;
 445 #endif
 446         mem_vtop_t mem_vtop;
 447         proc_t *p;
 448         pfn_t pfn = (pfn_t)PFN_INVALID;
 449         pid_t pid = 0;
 450         struct as *as;
 451         struct seg *seg;

 453         if (get_udatamodel() == DATAMODEL_NATIVE) {
 454                 if (copyin((void *)data, &mem_vtop, sizeof (mem_vtop_t)))
 455                         return (EFAULT);
 456         }
 457 #ifdef _SYSCALL32
 458         else {
 459                 if (copyin((void *)data, &vtop32, sizeof (mem_vtop32_t)))
 460                         return (EFAULT);
 461                 mem_vtop.m_as = (struct as *)(uintptr_t)vtop32.m_as;
 462                 mem_vtop.m_va = (void *)(uintptr_t)vtop32.m_va;

 464                 if (mem_vtop.m_as != NULL)
 465                         return (EINVAL);
 466         }
 467 #endif

 469         if (mem_vtop.m_as == &kas) {
 470                 pfn = hat_getpfnum(kas.a_hat, mem_vtop.m_va);
 471         } else {
 472                 if (mem_vtop.m_as == NULL) {
 473                         /*
 474                          * Assume the calling process's address space if the
 475                          * caller didn't specify one.
 476                          */
 477                         p = curthread->t_procp;
```



```
 478                         if (p == NULL)
 479                                 return (EIO);
 480                         mem_vtop.m_as = p->p_as;
 481                 }

 483                 mutex_enter(&pidlock);
 484                 for (p = practive; p != NULL; p = p->p_next) {
 485                         if (p->p_as == mem_vtop.m_as) {
 486                                 pid = p->p_pid;
 487                                 break;
 488                         }
 489                 }
 490                 mutex_exit(&pidlock);
 491                 if (p == NULL)
 492                         return (EIO);
 493                 p = sprlock(pid);
 494                 if (p == NULL)
 495                         return (EIO);
 496                 as = p->p_as;
 497                 if (as == mem_vtop.m_as) {
 498                         mutex_exit(&p->p_lock);
 499                         AS_LOCK_ENTER(as, RW_READER);
 499                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 500                         for (seg = AS_SEGFIRST(as); seg != NULL;
 501                             seg = AS_SEGNEXT(as, seg))
 502                                 if ((uintptr_t)mem_vtop.m_va -
 503                                     (uintptr_t)seg->s_base < seg->s_size)
 504                                         break;
 505                         if (seg != NULL)
 506                                 pfn = hat_getpfnum(as->a_hat, mem_vtop.m_va);
 507                         AS_LOCK_EXIT(as);
 507                         AS_LOCK_EXIT(as, &as->a_lock);
 508                         mutex_enter(&p->p_lock);
 509                 }
 510                 sprunlock(p);
 511         }
 512         mem_vtop.m_pfn = pfn;
 513         if (pfn == PFN_INVALID)
 514                 return (EIO);

 516         if (get_udatamodel() == DATAMODEL_NATIVE) {
 517                 if (copyout(&mem_vtop, (void *)data, sizeof (mem_vtop_t)))
 518                         return (EFAULT);
 519         }
 520 #ifdef _SYSCALL32
 521         else {
 522                 vtop32.m_pfn = mem_vtop.m_pfn;
 523                 if (copyout(&vtop32, (void *)data, sizeof (mem_vtop32_t)))
 524                         return (EFAULT);
 525         }
 526 #endif

 528         return (0);
 529 }
______unchanged_portion_omitted_
```

****
24367 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/io/physmem.c
patch as-lock-macro-simplification
****
_____***unchanged_portion_omitted_***

```
 636 /*
 637  * If the page has been hashed into the physmem vnode, then just look it up
 638  * and return it via pl, otherwise return ENOMEM as the map ioctl has not
 639  * succeeded on the given page.
 640  */
 641 /*ARGSUSED*/
 642 static int
 643 physmem_getpage(struct vnode *vp, offset_t off, size_t len, uint_t *protp,
 644     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr, enum seg_rw rw,
 645     struct cred *cr, caller_context_t *ct)
 646 {
 647         page_t *pp;

 649         ASSERT(len == PAGESIZE);
 650         ASSERT(AS_READ_HELD(seg->s_as));
 650         ASSERT(AS_READ_HELD(seg->s_as, &seg->s_as->a_lock));

 652         /*
 653          * If the page is in the hash, then we successfully claimed this
 654          * page earlier, so return it to the caller.
 655          */
 656         pp = page_lookup(vp, off, SE_SHARED);
 657         if (pp != NULL) {
 658                 pl[0] = pp;
 659                 pl[1] = NULL;
 660                 *protp = PROT_ALL;
 661                 return (0);
 662         }
 663         return (ENOMEM);
 664 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   80129 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/os/dumpsubr.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1401 /*
1402  * Dump the <as, va, pfn> information for a given address space.
1403  * SEGOP_DUMP() will call dump_addpage() for each page in the segment.
1404  */
1405 static void
1406 dump_as(struct as *as)
1407 {
1408         struct seg *seg;

1410         AS_LOCK_ENTER(as, RW_READER);
1410         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1411         for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
1412                 if (seg->s_as != as)
1413                         break;
1414                 if (seg->s_ops == NULL)
1415                         continue;
1416                 SEGOP_DUMP(seg);
1417         }
1418         AS_LOCK_EXIT(as);
1418         AS_LOCK_EXIT(as, &as->a_lock);

1420         if (seg != NULL)
1421                 cmn_err(CE_WARN, "invalid segment %p in address space %p",
1422                     (void *)seg, (void *)as);
1423 }
______unchanged_portion_omitted_
```

```
**********************************************************
   52300 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/os/exec.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1144 /*
1145  * Map a section of an executable file into the user's
1146  * address space.
1147  */
1148 int
1149 execmap(struct vnode *vp, caddr_t addr, size_t len, size_t zfodlen,
1150     off_t offset, int prot, int page, uint_t szc)
1151 {
1152         int error = 0;
1153         off_t oldoffset;
1154         caddr_t zfodbase, oldaddr;
1155         size_t end, oldlen;
1156         size_t zfoddiff;
1157         label_t ljb;
1158         proc_t *p = ttoproc(curthread);

1160         oldaddr = addr;
1161         addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1162         if (len) {
1163                 oldlen = len;
1164                 len += ((size_t)oldaddr - (size_t)addr);
1165                 oldoffset = offset;
1166                 offset = (off_t)((uintptr_t)offset & PAGEMASK);
1167                 if (page) {
1168                         spgcnt_t  prefltmem, availm, npages;
1169                         int preread;
1170                         uint_t mflag = MAP_PRIVATE | MAP_FIXED;

1172                         if ((prot & (PROT_WRITE | PROT_EXEC)) == PROT_EXEC) {
1173                                 mflag |= MAP_TEXT;
1174                         } else {
1175                                 mflag |= MAP_INITDATA;
1176                         }

1178                         if (valid_usr_range(addr, len, prot, p->p_as,
1179                             p->p_as->a_userlimit) != RANGE_OKAY) {
1180                                 error = ENOMEM;
1181                                 goto bad;
1182                         }
1183                         if (error = VOP_MAP(vp, (offset_t)offset,
1184                             p->p_as, &addr, len, prot, PROT_ALL,
1185                             mflag, CRED(), NULL))
1186                                 goto bad;

1188                         /*
1189                          * If the segment can fit, then we prefault
1190                          * the entire segment in.  This is based on the
1191                          * model that says the best working set of a
1192                          * small program is all of its pages.
1193                          */
1194                         npages = (spgcnt_t)btopr(len);
1195                         prefltmem = freemem - desfree;
1196                         preread =
1197                             (npages < prefltmem && len < PGTHRESH) ? 1 : 0;

1199                         /*
1200                          * If we aren't prefaulting the segment,
1201                          * increment "deficit", if necessary to ensure
1202                          * that pages will become available when this
```



```
1203                          * process starts executing.
1204                          */
1205                         availm = freemem - lotsfree;
1206                         if (preread == 0 && npages > availm &&
1207                             deficit < lotsfree) {
1208                                 deficit += MIN((pgcnt_t)(npages - availm),
1209                                     lotsfree - deficit);
1210                         }

1212                         if (preread) {
1213                                 TRACE_2(TR_FAC_PROC, TR_EXECMAP_PREREAD,
1214                                     "execmap preread:freemem %d size %lu",
1215                                     freemem, len);
1216                                 (void) as_fault(p->p_as->a_hat, p->p_as,
1217                                     (caddr_t)addr, len, F_INVAL, S_READ);
1218                         }
1219                 } else {
1220                         if (valid_usr_range(addr, len, prot, p->p_as,
1221                             p->p_as->a_userlimit) != RANGE_OKAY) {
1222                                 error = ENOMEM;
1223                                 goto bad;
1224                         }

1226                         if (error = as_map(p->p_as, addr, len,
1227                             segvn_create, zfod_argsp))
1228                                 goto bad;
1229                         /*
1230                          * Read in the segment in one big chunk.
1231                          */
1232                         if (error = vn_rdwr(UIO_READ, vp, (caddr_t)oldaddr,
1233                             oldlen, (offset_t)oldoffset, UIO_USERSPACE, 0,
1234                             (rlim64_t)0, CRED(), (ssize_t *)0))
1235                                 goto bad;
1236                         /*
1237                          * Now set protections.
1238                          */
1239                         if (prot != PROT_ZFOD) {
1240                                 (void) as_setprot(p->p_as, (caddr_t)addr,
1241                                     len, prot);
1242                         }
1243                 }
1244         }

1246         if (zfodlen) {
1247                 struct as *as = curproc->p_as;
1248                 struct seg *seg;
1249                 uint_t zprot = 0;

1251                 end = (size_t)addr + len;
1252                 zfodbase = (caddr_t)roundup(end, PAGESIZE);
1253                 zfoddiff = (uintptr_t)zfodbase - end;
1254                 if (zfoddiff) {
1255                         /*
1256                          * Before we go to zero the remaining space on the last
1257                          * page, make sure we have write permission.
1258                          *
1259                          * Normal illumos binaries don't even hit the case
1260                          * where we have to change permission on the last page
1261                          * since their protection is typically either
1262                          *    PROT_USER | PROT_WRITE | PROT_READ
1263                          * or
1264                          *    PROT_ZFOD (same as PROT_ALL).
1265                          *
1266                          * We need to be careful how we zero-fill the last page
1267                          * if the segment protection does not include
1268                          * PROT_WRITE. Using as_setprot() can cause the VM
```

```
1269                         * segment code to call segvn_vpage(), which must
1270                         * allocate a page struct for each page in the segment.
1271                         * If we have a very large segment, this may fail, so
1272                         * we have to check for that, even though we ignore
1273                         * other return values from as_setprot.
1274                         */

1276                        AS_LOCK_ENTER(as, RW_READER);
1276                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1277                        seg = as_segat(curproc->p_as, (caddr_t)end);
1278                        if (seg != NULL)
1279                                SEGOP_GETPROT(seg, (caddr_t)end, zfoddiff - 1,
1280                                    &zprot);
1281                        AS_LOCK_EXIT(as);
1281                        AS_LOCK_EXIT(as, &as->a_lock);

1283                        if (seg != NULL && (zprot & PROT_WRITE) == 0) {
1284                                if (as_setprot(as, (caddr_t)end, zfoddiff - 1,
1285                                    zprot | PROT_WRITE) == ENOMEM) {
1286                                        error = ENOMEM;
1287                                        goto bad;
1288                                }
1289                        }

1291                        if (on_fault(&ljb)) {
1292                                no_fault();
1293                                if (seg != NULL && (zprot & PROT_WRITE) == 0)
1294                                        (void) as_setprot(as, (caddr_t)end,
1295                                            zfoddiff - 1, zprot);
1296                                error = EFAULT;
1297                                goto bad;
1298                        }
1299                        uzero((void *)end, zfoddiff);
1300                        no_fault();
1301                        if (seg != NULL && (zprot & PROT_WRITE) == 0)
1302                                (void) as_setprot(as, (caddr_t)end,
1303                                    zfoddiff - 1, zprot);
1304                }
1305                if (zfodlen > zfoddiff) {
1306                        struct segvn_crargs crargs =
1307                            SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);

1309                        zfodlen -= zfoddiff;
1310                        if (valid_usr_range(zfodbase, zfodlen, prot, p->p_as,
1311                            p->p_as->a_userlimit) != RANGE_OKAY) {
1312                                error = ENOMEM;
1313                                goto bad;
1314                        }
1315                        if (szc > 0) {
1316                                /*
1317                                 * ASSERT alignment because the mapelfexec()
1318                                 * caller for the szc > 0 case extended zfod
1319                                 * so it's end is pgsz aligned.
1320                                 */
1321                                size_t pgsz = page_get_pagesize(szc);
1322                                ASSERT(IS_P2ALIGNED(zfodbase + zfodlen, pgsz));

1324                                if (IS_P2ALIGNED(zfodbase, pgsz)) {
1325                                        crargs.szc = szc;
1326                                } else {
1327                                        crargs.szc = AS_MAP_HEAP;
1328                                }
1329                        } else {
1330                                crargs.szc = AS_MAP_NO_LPOOB;
1331                        }
1332                        if (error = as_map(p->p_as, (caddr_t)zfodbase,
```



```
1333                                zfodlen, segvn_create, &crargs))
1334                                    goto bad;
1335                            if (prot != PROT_ZFOD) {
1336                                    (void) as_setprot(p->p_as, (caddr_t)zfodbase,
1337                                        zfodlen, prot);
1338                            }
1339                    }
1340            }
1341            return (0);
1342    bad:
1343            return (error);
1344    }
_____unchanged_portion_omitted_
```

```
**********************************************************
   37065 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/os/fork.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 128 /* ARGSUSED */
 129 static int64_t
 130 cfork(int isvfork, int isfork1, int flags)
 131 {
 132         proc_t *p = ttoproc(curthread);
 133         struct as *as;
 134         proc_t *cp, **orphpp;
 135         klwp_t *clone;
 136         kthread_t *t;
 137         task_t *tk;
 138         rval_t  r;
 139         int error;
 140         int i;
 141         rctl_set_t *dup_set;
 142         rctl_alloc_gp_t *dup_gp;
 143         rctl_entity_p_t e;
 144         lwpdir_t *ldp;
 145         lwpent_t *lep;
 146         lwpent_t *clep;

 148         /*
 149          * Allow only these two flags.
 150          */
 151         if ((flags & ~(FORK_NOSIGCHLD | FORK_WAITPID)) != 0) {
 152                 error = EINVAL;
 153                 atomic_inc_32(&curproc->p_zone->zone_ffmisc);
 154                 goto forkerr;
 155         }

 157         /*
 158          * fork is not supported for the /proc agent lwp.
 159          */
 160         if (curthread == p->p_agenttp) {
 161                 error = ENOTSUP;
 162                 atomic_inc_32(&curproc->p_zone->zone_ffmisc);
 163                 goto forkerr;
 164         }

 166         if ((error = secpolicy_basic_fork(CRED())) != 0) {
 167                 atomic_inc_32(&p->p_zone->zone_ffmisc);
 168                 goto forkerr;
 169         }

 171         /*
 172          * If the calling lwp is doing a fork1() then the
 173          * other lwps in this process are not duplicated and
 174          * don't need to be held where their kernel stacks can be
 175          * cloned.  If doing forkall(), the process is held with
 176          * SHOLDFORK, so that the lwps are at a point where their
 177          * stacks can be copied which is on entry or exit from
 178          * the kernel.
 179          */
 180         if (!holdlwps(isfork1 ? SHOLDFORK1 : SHOLDFORK)) {
 181                 aston(curthread);
 182                 error = EINTR;
 183                 atomic_inc_32(&p->p_zone->zone_ffmisc);
 184                 goto forkerr;
 185         }
```



```
 187 #if defined(__sparc)
 188         /*
 189          * Ensure that the user stack is fully constructed
 190          * before creating the child process structure.
 191          */
 192         (void) flush_user_windows_to_stack(NULL);
 193 #endif

 195         mutex_enter(&p->p_lock);
 196         /*
 197          * If this is vfork(), cancel any suspend request we might
 198          * have gotten from some other thread via lwp_suspend().
 199          * Otherwise we could end up with a deadlock on return
 200          * from the vfork() in both the parent and the child.
 201          */
 202         if (isvfork)
 203                 curthread->t_proc_flag &= ~TP_HOLDLWP;
 204         /*
 205          * Prevent our resource set associations from being changed during fork.
 206          */
 207         pool_barrier_enter();
 208         mutex_exit(&p->p_lock);

 210         /*
 211          * Create a child proc struct. Place a VN_HOLD on appropriate vnodes.
 212          */
 213         if (getproc(&cp, 0, GETPROC_USER) < 0) {
 214                 mutex_enter(&p->p_lock);
 215                 pool_barrier_exit();
 216                 continuelwps(p);
 217                 mutex_exit(&p->p_lock);
 218                 error = EAGAIN;
 219                 goto forkerr;
 220         }

 222         TRACE_2(TR_FAC_PROC, TR_PROC_FORK, "proc_fork:cp %p p %p", cp, p);

 224         /*
 225          * Assign an address space to child
 226          */
 227         if (isvfork) {
 228                 /*
 229                  * Clear any watched areas and remember the
 230                  * watched pages for restoring in vfwait().
 231                  */
 232                 as = p->p_as;
 233                 if (avl_numnodes(&as->a_wpage) != 0) {
 234                         AS_LOCK_ENTER(as, RW_WRITER);
 234                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 235                         as_clearwatch(as);
 236                         p->p_wpage = as->a_wpage;
 237                         avl_create(&as->a_wpage, wp_compare,
 238                             sizeof (struct watched_page),
 239                             offsetof(struct watched_page, wp_link));
 240                         AS_LOCK_EXIT(as);
 240                         AS_LOCK_EXIT(as, &as->a_lock);
 241                 }
 242                 cp->p_as = as;
 243                 cp->p_flag |= SVFORK;

 245                 /*
 246                  * Use the parent's shm segment list information for
 247                  * the child as it uses its address space till it execs.
 248                  */
 249                 cp->p_segacct = p->p_segacct;
 250         } else {
```

```
251                 /*
252                  * We need to hold P_PR_LOCK until the address space has
253                  * been duplicated and we've had a chance to remove from the
254                  * child any DTrace probes that were in the parent. Holding
255                  * P_PR_LOCK prevents any new probes from being added and any
256                  * extant probes from being removed.
257                  */
258                 mutex_enter(&p->p_lock);
259                 sprlock_proc(p);
260                 p->p_flag |= SFORKING;
261                 mutex_exit(&p->p_lock);

263                 error = as_dup(p->p_as, cp);
264                 if (error != 0) {
265                         mutex_enter(&p->p_lock);
266                         sprunlock(p);
267                         fork_fail(cp);
268                         mutex_enter(&pidlock);
269                         orphpp = &p->p_orphan;
270                         while (*orphpp != cp)
271                                 orphpp = &(*orphpp)->p_nextorph;
272                         *orphpp = cp->p_nextorph;
273                         if (p->p_child == cp)
274                                 p->p_child = cp->p_sibling;
275                         if (cp->p_sibling)
276                                 cp->p_sibling->p_psibling = cp->p_psibling;
277                         if (cp->p_psibling)
278                                 cp->p_psibling->p_sibling = cp->p_sibling;
279                         mutex_enter(&cp->p_lock);
280                         tk = cp->p_task;
281                         task_detach(cp);
282                         ASSERT(cp->p_pool->pool_ref > 0);
283                         atomic_dec_32(&cp->p_pool->pool_ref);
284                         mutex_exit(&cp->p_lock);
285                         pid_exit(cp, tk);
286                         mutex_exit(&pidlock);
287                         task_rele(tk);

289                         mutex_enter(&p->p_lock);
290                         p->p_flag &= ~SFORKING;
291                         pool_barrier_exit();
292                         continuelwps(p);
293                         mutex_exit(&p->p_lock);
294                         /*
295                          * Preserve ENOMEM error condition but
296                          * map all others to EAGAIN.
297                          */
298                         error = (error == ENOMEM) ? ENOMEM : EAGAIN;
299                         atomic_inc_32(&p->p_zone->zone_ffnomem);
300                         goto forkerr;
301                 }

303                 /*
304                  * Remove all DTrace tracepoints from the child process. We
305                  * need to do this _before_ duplicating USDT providers since
306                  * any associated probes may be immediately enabled.
307                  */
308                 if (p->p_dtrace_count > 0)
309                         dtrace_fasttrap_fork(p, cp);

311                 mutex_enter(&p->p_lock);
312                 sprunlock(p);

314                 /* Duplicate parent's shared memory */
315                 if (p->p_segacct)
316                         shmfork(p, cp);
```



```
318                 /*
319                  * Duplicate any helper actions and providers. The SFORKING
320                  * we set above informs the code to enable USDT probes that
321                  * sprlock() may fail because the child is being forked.
322                  */
323                 if (p->p_dtrace_helpers != NULL) {
324                         ASSERT(dtrace_helpers_fork != NULL);
325                         (*dtrace_helpers_fork)(p, cp);
326                 }

328                 mutex_enter(&p->p_lock);
329                 p->p_flag &= ~SFORKING;
330                 mutex_exit(&p->p_lock);
331         }

333         /*
334          * Duplicate parent's resource controls.
335          */
336         dup_set = rctl_set_create();
337         for (;;) {
338                 dup_gp = rctl_set_dup_prealloc(p->p_rctls);
339                 mutex_enter(&p->p_rctls->rcs_lock);
340                 if (rctl_set_dup_ready(p->p_rctls, dup_gp))
341                         break;
342                 mutex_exit(&p->p_rctls->rcs_lock);
343                 rctl_prealloc_destroy(dup_gp);
344         }
345         e.rcep_p.proc = cp;
346         e.rcep_t = RCENTITY_PROCESS;
347         cp->p_rctls = rctl_set_dup(p->p_rctls, p, cp, &e, dup_set, dup_gp,
348             RCD_DUP | RCD_CALLBACK);
349         mutex_exit(&p->p_rctls->rcs_lock);

351         rctl_prealloc_destroy(dup_gp);

353         /*
354          * Allocate the child's lwp directory and lwpid hash table.
355          */
356         if (isfork1)
357                 cp->p_lwpdir_sz = 2;
358         else
359                 cp->p_lwpdir_sz = p->p_lwpdir_sz;
360         cp->p_lwpdir = cp->p_lwpfree = ldp =
361             kmem_zalloc(cp->p_lwpdir_sz * sizeof (lwpdir_t), KM_SLEEP);
362         for (i = 1; i < cp->p_lwpdir_sz; i++, ldp++)
363                 ldp->ld_next = ldp + 1;
364         cp->p_tidhash_sz = (cp->p_lwpdir_sz + 2) / 2;
365         cp->p_tidhash =
366             kmem_zalloc(cp->p_tidhash_sz * sizeof (tidhash_t), KM_SLEEP);

368         /*
369          * Duplicate parent's lwps.
370          * Mutual exclusion is not needed because the process is
371          * in the hold state and only the current lwp is running.
372          */
373         klgrpset_clear(cp->p_lgrpset);
374         if (isfork1) {
375                 clone = forklwp(ttolwp(curthread), cp, curthread->t_tid);
376                 if (clone == NULL)
377                         goto forklwperr;
378                 /*
379                  * Inherit only the lwp_wait()able flag,
380                  * Daemon threads should not call fork1(), but oh well...
381                  */
382                 lwptot(clone)->t_proc_flag |=
```

```
383                     (curthread->t_proc_flag & TP_TWAIT);
384         } else {
385                 /* this is forkall(), no one can be in lwp_wait() */
386                 ASSERT(p->p_lwpwait == 0 && p->p_lwpdwait == 0);
387                 /* for each entry in the parent's lwp directory... */
388                 for (i = 0, ldp = p->p_lwpdir; i < p->p_lwpdir_sz; i++, ldp++) {
389                         klwp_t *clwp;
390                         kthread_t *ct;

392                         if ((lep = ldp->ld_entry) == NULL)
393                                 continue;

395                         if ((t = lep->le_thread) != NULL) {
396                                 clwp = forklwp(ttolwp(t), cp, t->t_tid);
397                                 if (clwp == NULL)
398                                         goto forklwperr;
399                                 ct = lwptot(clwp);
400                                 /*
401                                  * Inherit lwp_wait()able and daemon flags.
402                                  */
403                                 ct->t_proc_flag |=
404                                     (t->t_proc_flag & (TP_TWAIT|TP_DAEMON));
405                                 /*
406                                  * Keep track of the clone of curthread to
407                                  * post return values through lwp_setrval().
408                                  * Mark other threads for special treatment
409                                  * by lwp_rtt() / post_syscall().
410                                  */
411                                 if (t == curthread)
412                                         clone = clwp;
413                                 else
414                                         ct->t_flag |= T_FORKALL;
415                         } else {
416                                 /*
417                                  * Replicate zombie lwps in the child.
418                                  */
419                                 clep = kmem_zalloc(sizeof (*clep), KM_SLEEP);
420                                 clep->le_lwpid = lep->le_lwpid;
421                                 clep->le_start = lep->le_start;
422                                 lwp_hash_in(cp, clep,
423                                     cp->p_tidhash, cp->p_tidhash_sz, 0);
424                         }
425                 }
426         }

428         /*
429          * Put new process in the parent's process contract, or put it
430          * in a new one if there is an active process template.  Send a
431          * fork event (if requested) to whatever contract the child is
432          * a member of.  Fails if the parent has been SIGKILLed.
433          */
434         if (contract_process_fork(NULL, cp, p, B_TRUE) == NULL) {
435                 atomic_inc_32(&p->p_zone->zone_ffmisc);
436                 goto forklwperr;
437         }

439         /*
440          * No fork failures occur beyond this point.
441          */

443         cp->p_lwpid = p->p_lwpid;
444         if (!isfork1) {
445                 cp->p_lwpdaemon = p->p_lwpdaemon;
446                 cp->p_zombcnt = p->p_zombcnt;
447                 /*
448                  * If the parent's lwp ids have wrapped around, so have the
```



```
449                  * child's.
450                  */
451                 cp->p_flag |= p->p_flag & SLWPWRAP;
452         }

454         mutex_enter(&p->p_lock);
455         corectl_path_hold(cp->p_corefile = p->p_corefile);
456         corectl_content_hold(cp->p_content = p->p_content);
457         mutex_exit(&p->p_lock);

459         /*
460          * Duplicate process context ops, if any.
461          */
462         if (p->p_pctx)
463                 forkpctx(p, cp);

465 #ifdef __sparc
466         utrap_dup(p, cp);
467 #endif
468         /*
469          * If the child process has been marked to stop on exit
470          * from this fork, arrange for all other lwps to stop in
471          * sympathy with the active lwp.
472          */
473         if (PTOU(cp)->u_systrap &&
474             prismember(&PTOU(cp)->u_exitmask, curthread->t_sysnum)) {
475                 mutex_enter(&cp->p_lock);
476                 t = cp->p_tlist;
477                 do {
478                         t->t_proc_flag |= TP_PRSTOP;
479                         aston(t);       /* so TP_PRSTOP will be seen */
480                 } while ((t = t->t_forw) != cp->p_tlist);
481                 mutex_exit(&cp->p_lock);
482         }
483         /*
484          * If the parent process has been marked to stop on exit
485          * from this fork, and its asynchronous-stop flag has not
486          * been set, arrange for all other lwps to stop before
487          * they return back to user level.
488          */
489         if (!(p->p_proc_flag & P_PR_ASYNC) && PTOU(p)->u_systrap &&
490             prismember(&PTOU(p)->u_exitmask, curthread->t_sysnum)) {
491                 mutex_enter(&p->p_lock);
492                 t = p->p_tlist;
493                 do {
494                         t->t_proc_flag |= TP_PRSTOP;
495                         aston(t);       /* so TP_PRSTOP will be seen */
496                 } while ((t = t->t_forw) != p->p_tlist);
497                 mutex_exit(&p->p_lock);
498         }

500         if (PROC_IS_BRANDED(p))
501                 BROP(p)->b_lwp_setrval(clone, p->p_pid, 1);
502         else
503                 lwp_setrval(clone, p->p_pid, 1);

505         /* set return values for parent */
506         r.r_val1 = (int)cp->p_pid;
507         r.r_val2 = 0;

509         /*
510          * pool_barrier_exit() can now be called because the child process has:
511          * - all identifying features cloned or set (p_pid, p_task, p_pool)
512          * - all resource sets associated (p_tlist->*->t_cpupart, p_as->a_mset)
513          * - any other fields set which are used in resource set binding.
514          */
```

```
515         mutex_enter(&p->p_lock);
516         pool_barrier_exit();
517         mutex_exit(&p->p_lock);

519         mutex_enter(&pidlock);
520         mutex_enter(&cp->p_lock);

522         /*
523          * Set flags telling the child what (not) to do on exit.
524          */
525         if (flags & FORK_NOSIGCHLD)
526                 cp->p_pidflag |= CLDNOSIGCHLD;
527         if (flags & FORK_WAITPID)
528                 cp->p_pidflag |= CLDWAITPID;

530         /*
531          * Now that there are lwps and threads attached, add the new
532          * process to the process group.
533          */
534         pgjoin(cp, p->p_pgidp);
535         cp->p_stat = SRUN;
536         /*
537          * We are now done with all the lwps in the child process.
538          */
539         t = cp->p_tlist;
540         do {
541                 /*
542                  * Set the lwp_suspend()ed lwps running.
543                  * They will suspend properly at syscall exit.
544                  */
545                 if (t->t_proc_flag & TP_HOLDLWP)
546                         lwp_create_done(t);
547                 else {
548                         /* set TS_CREATE to allow continuelwps() to work */
549                         thread_lock(t);
550                         ASSERT(t->t_state == TS_STOPPED &&
551                             !(t->t_schedflag & (TS_CREATE|TS_CSTART)));
552                         t->t_schedflag |= TS_CREATE;
553                         thread_unlock(t);
554                 }
555         } while ((t = t->t_forw) != cp->p_tlist);
556         mutex_exit(&cp->p_lock);

558         if (isvfork) {
559                 CPU_STATS_ADDQ(CPU, sys, sysvfork, 1);
560                 mutex_enter(&p->p_lock);
561                 p->p_flag |= SVFWAIT;
562                 curthread->t_flag |= T_VFPARENT;
563                 DTRACE_PROC1(create, proc_t *, cp);
564                 cv_broadcast(&pr_pid_cv[p->p_slot]);    /* inform /proc */
565                 mutex_exit(&p->p_lock);
566                 /*
567                  * Grab child's p_lock before dropping pidlock to ensure
568                  * the process will not disappear before we set it running.
569                  */
570                 mutex_enter(&cp->p_lock);
571                 mutex_exit(&pidlock);
572                 sigdefault(cp);
573                 continuelwps(cp);
574                 mutex_exit(&cp->p_lock);
575         } else {
576                 CPU_STATS_ADDQ(CPU, sys, sysfork, 1);
577                 DTRACE_PROC1(create, proc_t *, cp);
578                 /*
579                  * It is CL_FORKRET's job to drop pidlock.
580                  * If we do it here, the process could be set running
```



```
581                  * and disappear before CL_FORKRET() is called.
582                  */
583                 CL_FORKRET(curthread, cp->p_tlist);
584                 schedctl_set_cidpri(curthread);
585                 ASSERT(MUTEX_NOT_HELD(&pidlock));
586         }

588         return (r.r_vals);

590 forklwperr:
591         if (isvfork) {
592                 if (avl_numnodes(&p->p_wpage) != 0) {
593                         /* restore watchpoints to parent */
594                         as = p->p_as;
595                         AS_LOCK_ENTER(as, RW_WRITER);
595                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
596                         as->a_wpage = p->p_wpage;
597                         avl_create(&p->p_wpage, wp_compare,
598                             sizeof (struct watched_page),
599                             offsetof(struct watched_page, wp_link));
600                         as_setwatch(as);
601                         AS_LOCK_EXIT(as);
601                         AS_LOCK_EXIT(as, &as->a_lock);
602                 }
603         } else {
604                 if (cp->p_segacct)
605                         shmexit(cp);
606                 as = cp->p_as;
607                 cp->p_as = &kas;
608                 as_free(as);
609         }

611         if (cp->p_lwpdir) {
612                 for (i = 0, ldp = cp->p_lwpdir; i < cp->p_lwpdir_sz; i++, ldp++)
613                         if ((lep = ldp->ld_entry) != NULL)
614                                 kmem_free(lep, sizeof (*lep));
615                 kmem_free(cp->p_lwpdir,
616                     cp->p_lwpdir_sz * sizeof (*cp->p_lwpdir));
617         }
618         cp->p_lwpdir = NULL;
619         cp->p_lwpfree = NULL;
620         cp->p_lwpdir_sz = 0;

622         if (cp->p_tidhash)
623                 kmem_free(cp->p_tidhash,
624                     cp->p_tidhash_sz * sizeof (*cp->p_tidhash));
625         cp->p_tidhash = NULL;
626         cp->p_tidhash_sz = 0;

628         forklwp_fail(cp);
629         fork_fail(cp);
630         rctl_set_free(cp->p_rctls);
631         mutex_enter(&pidlock);

633         /*
634          * Detach failed child from task.
635          */
636         mutex_enter(&cp->p_lock);
637         tk = cp->p_task;
638         task_detach(cp);
639         ASSERT(cp->p_pool->pool_ref > 0);
640         atomic_dec_32(&cp->p_pool->pool_ref);
641         mutex_exit(&cp->p_lock);

643         orphpp = &p->p_orphan;
644         while (*orphpp != cp)
```

```
 645                orphpp = &(*orphpp)->p_nextorph;
 646        *orphpp = cp->p_nextorph;
 647        if (p->p_child == cp)
 648                p->p_child = cp->p_sibling;
 649        if (cp->p_sibling)
 650                cp->p_sibling->p_psibling = cp->p_psibling;
 651        if (cp->p_psibling)
 652                cp->p_psibling->p_sibling = cp->p_sibling;
 653        pid_exit(cp, tk);
 654        mutex_exit(&pidlock);

 656        task_rele(tk);

 658        mutex_enter(&p->p_lock);
 659        pool_barrier_exit();
 660        continuelwps(p);
 661        mutex_exit(&p->p_lock);
 662        error = EAGAIN;
 663 forkerr:
 664        return ((int64_t)set_errno(error));
 665 }
______unchanged_portion_omitted_

1385 /*
1386  * Wait for child to exec or exit.
1387  * Called by parent of vfork'ed process.
1388  * See important comments in relvm(), above.
1389  */
1390 void
1391 vfwait(pid_t pid)
1392 {
1393        int signalled = 0;
1394        proc_t *pp = ttoproc(curthread);
1395        proc_t *cp;

1397        /*
1398         * Wait for child to exec or exit.
1399         */
1400        for (;;) {
1401                mutex_enter(&pidlock);
1402                cp = prfind(pid);
1403                if (cp == NULL || cp->p_parent != pp) {
1404                        /*
1405                         * Child has exit()ed.
1406                         */
1407                        mutex_exit(&pidlock);
1408                        break;
1409                }
1410                /*
1411                 * Grab the child's p_lock before releasing pidlock.
1412                 * Otherwise, the child could exit and we would be
1413                 * referencing invalid memory.
1414                 */
1415                mutex_enter(&cp->p_lock);
1416                mutex_exit(&pidlock);
1417                if (!(cp->p_flag & SVFORK)) {
1418                        /*
1419                         * Child has exec()ed or is exit()ing.
1420                         */
1421                        mutex_exit(&cp->p_lock);
1422                        break;
1423                }
1424                mutex_enter(&pp->p_lock);
1425                mutex_exit(&cp->p_lock);
1426                /*
1427                 * We might be waked up spuriously from the cv_wait().
```



```
1428                 * We have to do the whole operation over again to be
1429                 * sure the child's SVFORK flag really is turned off.
1430                 * We cannot make reference to the child because it can
1431                 * exit before we return and we would be referencing
1432                 * invalid memory.
1433                 *
1434                 * Because this is potentially a very long-term wait,
1435                 * we call cv_wait_sig() (for its jobcontrol and /proc
1436                 * side-effects) unless there is a current signal, in
1437                 * which case we use cv_wait() because we cannot return
1438                 * from this function until the child has released the
1439                 * address space.  Calling cv_wait_sig() with a current
1440                 * signal would lead to an indefinite loop here because
1441                 * cv_wait_sig() returns immediately in this case.
1442                 */
1443                if (signalled)
1444                        cv_wait(&pp->p_cv, &pp->p_lock);
1445                else
1446                        signalled = !cv_wait_sig(&pp->p_cv, &pp->p_lock);
1447                mutex_exit(&pp->p_lock);
1448        }

1450        /* restore watchpoints to parent */
1451        if (pr_watch_active(pp)) {
1452                struct as *as = pp->p_as;
1453                AS_LOCK_ENTER(as, RW_WRITER);
1453                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1454                as_setwatch(as);
1455                AS_LOCK_EXIT(as);
1455                AS_LOCK_EXIT(as, &as->a_lock);
1456        }

1458        mutex_enter(&pp->p_lock);
1459        prbarrier(pp);  /* barrier against /proc locking */
1460        continuelwps(pp);
1461        mutex_exit(&pp->p_lock);
1462 }
______unchanged_portion_omitted_
```

```
**********************************************************
   69021 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/os/mmapobj.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1434 /*
1435  * Check the address space to see if the virtual addresses to be used are
1436  * available.  If they are not, return errno for failure.  On success, 0
1437  * will be returned, and the virtual addresses for each mmapobj_result_t
1438  * will be reserved.  Note that a reservation could have earlier been made
1439  * for a given segment via a /dev/null mapping.  If that is the case, then
1440  * we can use that VA space for our mappings.
1441  * Note: this function will only be used for ET_EXEC binaries.
1442  */
1443 int
1444 check_exec_addrs(int loadable, mmapobj_result_t *mrp, caddr_t start_addr)
1445 {
1446         int i;
1447         struct as *as = curproc->p_as;
1448         struct segvn_crargs crargs = SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
1449         int ret;
1450         caddr_t myaddr;
1451         size_t mylen;
1452         struct seg *seg;

1454         /* No need to reserve swap space now since it will be reserved later */
1455         crargs.flags |= MAP_NORESERVE;
1456         as_rangelock(as);
1457         for (i = 0; i < loadable; i++) {

1459                 myaddr = start_addr + (size_t)mrp[i].mr_addr;
1460                 mylen = mrp[i].mr_msize;

1462                 /* See if there is a hole in the as for this range */
1463                 if (as_gap(as, mylen, &myaddr, &mylen, 0, NULL) == 0) {
1464                         ASSERT(myaddr == start_addr + (size_t)mrp[i].mr_addr);
1465                         ASSERT(mylen == mrp[i].mr_msize);

1467 #ifdef DEBUG
1468                         if (MR_GET_TYPE(mrp[i].mr_flags) == MR_PADDING) {
1469                                 MOBJ_STAT_ADD(exec_padding);
1470                         }
1471 #endif
1472                         ret = as_map(as, myaddr, mylen, segvn_create, &crargs);
1473                         if (ret) {
1474                                 as_rangeunlock(as);
1475                                 mmapobj_unmap_exec(mrp, i, start_addr);
1476                                 return (ret);
1477                         }
1478                 } else {
1479                         /*
1480                          * There is a mapping that exists in the range
1481                          * so check to see if it was a "reservation"
1482                          * from /dev/null.  The mapping is from
1483                          * /dev/null if the mapping comes from
1484                          * segdev and the type is neither MAP_SHARED
1485                          * nor MAP_PRIVATE.
1486                          */
1487                         AS_LOCK_ENTER(as, RW_READER);
1487                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1488                         seg = as_findseg(as, myaddr, 0);
1489                         MOBJ_STAT_ADD(exec_addr_mapped);
1490                         if (seg && seg->s_ops == &segdev_ops &&
1491                             ((SEGOP_GETTYPE(seg, myaddr) &
```



```
1492                             (MAP_SHARED | MAP_PRIVATE)) == 0) &&
1493                             myaddr >= seg->s_base &&
1494                             myaddr + mylen <=
1495                             seg->s_base + seg->s_size) {
1496                                 MOBJ_STAT_ADD(exec_addr_devnull);
1497                                 AS_LOCK_EXIT(as);
1497                                 AS_LOCK_EXIT(as, &as->a_lock);
1498                                 (void) as_unmap(as, myaddr, mylen);
1499                                 ret = as_map(as, myaddr, mylen, segvn_create,
1500                                     &crargs);
1501                                 mrp[i].mr_flags |= MR_RESV;
1502                                 if (ret) {
1503                                         as_rangeunlock(as);
1504                                         /* Need to remap what we unmapped */
1505                                         mmapobj_unmap_exec(mrp, i + 1,
1506                                             start_addr);
1507                                         return (ret);
1508                                 }
1509                         } else {
1510                                 AS_LOCK_EXIT(as);
1510                                 AS_LOCK_EXIT(as, &as->a_lock);
1511                                 as_rangeunlock(as);
1512                                 mmapobj_unmap_exec(mrp, i, start_addr);
1513                                 MOBJ_STAT_ADD(exec_addr_in_use);
1514                                 return (EADDRINUSE);
1515                         }
1516                 }
1517         }
1518         as_rangeunlock(as);
1519         return (0);
1520 }
______unchanged_portion_omitted_
```

```
**********************************************************
   18066 Wed Nov 25 13:59:37 2015
new/usr/src/uts/common/os/move.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 570 /*
 571  * Initialize a uioa_t for a given uio_t for the current user context,
 572  * copy the common uio_t to the uioa_t, walk the shared iovec_t and
 573  * lock down the user-land page(s) containing iovec_t data, then mapin
 574  * user-land pages using segkpm.
 575  */
 576 int
 577 uioainit(uio_t *uiop, uioa_t *uioap)
 578 {
 579         caddr_t addr;
 580         page_t          **pages;
 581         int             off;
 582         int             len;
 583         proc_t          *procp = ttoproc(curthread);
 584         struct as       *as = procp->p_as;
 585         iovec_t         *iov = uiop->uio_iov;
 586         int32_t         iovcnt = uiop->uio_iovcnt;
 587         uioa_page_t     *locked = uioap->uioa_locked;
 588         dcopy_handle_t  channel;
 589         int             error;

 591         if (! (uioap->uioa_state & UIOA_ALLOC)) {
 592                 /* Can only init() a freshly allocated uioa_t */
 593                 return (EINVAL);
 594         }

 596         error = dcopy_alloc(DCOPY_NOSLEEP, &channel);
 597         if (error == DCOPY_NORESOURCES) {
 598                 /* Turn off uioa */
 599                 uioasync.enabled = B_FALSE;
 600                 return (ENODEV);
 601         }
 602         if (error != DCOPY_SUCCESS) {
 603                 /* Alloc failed */
 604                 return (EIO);
 605         }

 607         uioap->uioa_hwst[UIO_DCOPY_CHANNEL] = channel;
 608         uioap->uioa_hwst[UIO_DCOPY_CMD] = NULL;

 610         /* Indicate uioa_t (will be) initialized */
 611         uioap->uioa_state = UIOA_INIT;

 613         uioap->uioa_mbytes = 0;

 615         /* uio_t/uioa_t uio_t common struct copy */
 616         *((uio_t *)uioap) = *uiop;

 618         /* initialize *uiop->uio_iov */
 619         if (iovcnt > UIOA_IOV_MAX) {
 620                 /* Too big? */
 621                 return (E2BIG);
 622         }
 623         uioap->uio_iov = iov;
 624         uioap->uio_iovcnt = iovcnt;

 626         /* Mark the uioap as such */
 627         uioap->uio_extflg |= UIO_ASYNC;
```



```
 629         /*
 630          * For each iovec_t, lock-down the page(s) backing the iovec_t
 631          * and save the page_t list for phys addr use in uioamove().
 632          */
 633         iov = uiop->uio_iov;
 634         iovcnt = uiop->uio_iovcnt;
 635         while (iovcnt > 0) {
 636                 addr = iov->iov_base;
 637                 off = (uintptr_t)addr & PAGEOFFSET;
 638                 addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
 639                 len = iov->iov_len + off;

 641                 /* Lock down page(s) for the iov span */
 642                 if ((error = as_pagelock(as, &pages,
 643                     iov->iov_base, iov->iov_len, S_WRITE)) != 0) {
 644                         /* Error */
 645                         goto cleanup;
 646                 }

 648                 if (pages == NULL) {
 649                         /*
 650                          * Need page_t list, really only need
 651                          * a pfn list so build one.
 652                          */
 653                         pfn_t   *pfnp;
 654                         int     pcnt = len >> PAGESHIFT;

 656                         if (off)
 657                                 pcnt++;
 658                         if ((pfnp = kmem_alloc(pcnt * sizeof (pfnp),
 659                             KM_NOSLEEP)) == NULL) {
 660                                 error = ENOMEM;
 661                                 goto cleanup;
 662                         }
 663                         locked->uioa_ppp = (void **)pfnp;
 664                         locked->uioa_pfncnt = pcnt;
 665                         AS_LOCK_ENTER(as, RW_READER);
 665                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 666                         while (pcnt-- > 0) {
 667                                 *pfnp++ = hat_getpfnum(as->a_hat, addr);
 668                                 addr += PAGESIZE;
 669                         }
 670                         AS_LOCK_EXIT(as);
 670                         AS_LOCK_EXIT(as, &as->a_lock);
 671                 } else {
 672                         /* Have a page_t list, save it */
 673                         locked->uioa_ppp = (void **)pages;
 674                         locked->uioa_pfncnt = 0;
 675                 }
 676                 /* Save for as_pageunlock() in uioafini() */
 677                 locked->uioa_base = iov->iov_base;
 678                 locked->uioa_len = iov->iov_len;
 679                 locked++;

 681                 /* Next iovec_t */
 682                 iov++;
 683                 iovcnt--;
 684         }
 685         /* Initialize curret pointer into uioa_locked[] and it's uioa_ppp */
 686         uioap->uioa_lcur = uioap->uioa_locked;
 687         uioap->uioa_lppp = uioap->uioa_lcur->uioa_ppp;
 688         return (0);

 690 cleanup:
 691         /* Unlock any previously locked page_t(s) */
 692         while (locked > uioap->uioa_locked) {
```

```
 693                 locked--;
 694                 as_pageunlock(as, (page_t **)locked->uioa_ppp,
 695                     locked->uioa_base, locked->uioa_len, S_WRITE);
 696         }

 698         /* Last indicate uioa_t still in alloc state */
 699         uioap->uioa_state = UIOA_ALLOC;
 700         uioap->uioa_mbytes = 0;

 702         return (error);
 703 }
```

unchanged_portion_omitted_

```
**********************************************************
  248802 Wed Nov 25 13:59:38 2015
new/usr/src/uts/common/os/sunddi.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

8219 /*
8220  * A consolidation private function which is essentially equivalent to
8221  * ddi_umem_lock but with the addition of arguments ops_vector and procp.
8222  * A call to as_add_callback is done if DDI_UMEMLOCK_LONGTERM is set, and
8223  * the ops_vector is valid.
8224  *
8225  * Lock the virtual address range in the current process and create a
8226  * ddi_umem_cookie (of type UMEM_LOCKED). This can be used to pass to
8227  * ddi_umem_iosetup to create a buf or do devmap_umem_setup/remap to export
8228  * to user space.
8229  *
8230  * Note: The resource control accounting currently uses a full charge model
8231  * in other words attempts to lock the same/overlapping areas of memory
8232  * will deduct the full size of the buffer from the projects running
8233  * counter for the device locked memory.
8234  *
8235  * addr, size should be PAGESIZE aligned
8236  *
8237  * flags - DDI_UMEMLOCK_READ, DDI_UMEMLOCK_WRITE or both
8238  *      identifies whether the locked memory will be read or written or both
8239  *      DDI_UMEMLOCK_LONGTERM  must be set when the locking will
8240  * be maintained for an indefinitely long period (essentially permanent),
8241  * rather than for what would be required for a typical I/O completion.
8242  * When DDI_UMEMLOCK_LONGTERM is set, umem_lockmemory will return EFAULT
8243  * if the memory pertains to a regular file which is mapped MAP_SHARED.
8244  * This is to prevent a deadlock if a file truncation is attempted after
8245  * after the locking is done.
8246  *
8247  * Returns 0 on success
8248  *      EINVAL - for invalid parameters
8249  *      EPERM, ENOMEM and other error codes returned by as_pagelock
8250  *      ENOMEM - is returned if the current request to lock memory exceeds
8251  *              *.max-locked-memory resource control value.
8252  *      EFAULT - memory pertains to a regular file mapped shared and
8253  *              and DDI_UMEMLOCK_LONGTERM flag is set
8254  *      EAGAIN - could not start the ddi_umem_unlock list processing thread
8255  */
8256 int
8257 umem_lockmemory(caddr_t addr, size_t len, int flags, ddi_umem_cookie_t *cookie,
8258                 struct umem_callback_ops *ops_vector,
8259                 proc_t *procp)
8260 {
8261         int     error;
8262         struct ddi_umem_cookie *p;
8263         void    (*driver_callback)() = NULL;
8264         struct as *as;
8265         struct seg              *seg;
8266         vnode_t                 *vp;

8268         /* Allow device drivers to not have to reference "curproc" */
8269         if (procp == NULL)
8270                 procp = curproc;
8271         as = procp->p_as;
8272         *cookie = NULL;         /* in case of any error return */

8274         /* These are the only three valid flags */
8275         if ((flags & ~(DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE |
8276             DDI_UMEMLOCK_LONGTERM)) != 0)
8277                 return (EINVAL);
```



```
8279         /* At least one (can be both) of the two access flags must be set */
8280         if ((flags & (DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE)) == 0)
8281                 return (EINVAL);

8283         /* addr and len must be page-aligned */
8284         if (((uintptr_t)addr & PAGEOFFSET) != 0)
8285                 return (EINVAL);

8287         if ((len & PAGEOFFSET) != 0)
8288                 return (EINVAL);

8290         /*
8291          * For longterm locking a driver callback must be specified; if
8292          * not longterm then a callback is optional.
8293          */
8294         if (ops_vector != NULL) {
8295                 if (ops_vector->cbo_umem_callback_version !=
8296                     UMEM_CALLBACK_VERSION)
8297                         return (EINVAL);
8298                 else
8299                         driver_callback = ops_vector->cbo_umem_lock_cleanup;
8300         }
8301         if ((driver_callback == NULL) && (flags & DDI_UMEMLOCK_LONGTERM))
8302                 return (EINVAL);

8304         /*
8305          * Call i_ddi_umem_unlock_thread_start if necessary.  It will
8306          * be called on first ddi_umem_lock or umem_lockmemory call.
8307          */
8308         if (ddi_umem_unlock_thread == NULL)
8309                 i_ddi_umem_unlock_thread_start();

8311         /* Allocate memory for the cookie */
8312         p = kmem_zalloc(sizeof (struct ddi_umem_cookie), KM_SLEEP);

8314         /* Convert the flags to seg_rw type */
8315         if (flags & DDI_UMEMLOCK_WRITE) {
8316                 p->s_flags = S_WRITE;
8317         } else {
8318                 p->s_flags = S_READ;
8319         }

8321         /* Store procp in cookie for later iosetup/unlock */
8322         p->procp = (void *)procp;

8324         /*
8325          * Store the struct as pointer in cookie for later use by
8326          * ddi_umem_unlock.  The proc->p_as will be stale if ddi_umem_unlock
8327          * is called after relvm is called.
8328          */
8329         p->asp = as;

8331         /*
8332          * The size field is needed for lockmem accounting.
8333          */
8334         p->size = len;
8335         init_lockedmem_rctl_flag(p);

8337         if (umem_incr_devlockmem(p) != 0) {
8338                 /*
8339                  * The requested memory cannot be locked
8340                  */
8341                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8342                 *cookie = (ddi_umem_cookie_t)NULL;
8343                 return (ENOMEM);
```

```
8344         }

8346         /* Lock the pages corresponding to addr, len in memory */
8347         error = as_pagelock(as, &(p->pparray), addr, len, p->s_flags);
8348         if (error != 0) {
8349                 umem_decr_devlockmem(p);
8350                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8351                 *cookie = (ddi_umem_cookie_t)NULL;
8352                 return (error);
8353         }

8355         /*
8356          * For longterm locking the addr must pertain to a seg_vn segment or
8357          * or a seg_spt segment.
8358          * If the segment pertains to a regular file, it cannot be
8359          * mapped MAP_SHARED.
8360          * This is to prevent a deadlock if a file truncation is attempted
8361          * after the locking is done.
8362          * Doing this after as_pagelock guarantees persistence of the as; if
8363          * an unacceptable segment is found, the cleanup includes calling
8364          * as_pageunlock before returning EFAULT.
8365          *
8366          * segdev is allowed here as it is already locked.  This allows
8367          * for memory exported by drivers through mmap() (which is already
8368          * locked) to be allowed for LONGTERM.
8369          */
8370         if (flags & DDI_UMEMLOCK_LONGTERM) {
8371                 extern  struct seg_ops segspt_shmops;
8372                 extern  struct seg_ops segdev_ops;
8373                 AS_LOCK_ENTER(as, RW_READER);
8373                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
8374                 for (seg = as_segat(as, addr); ; seg = AS_SEGNEXT(as, seg)) {
8375                         if (seg == NULL || seg->s_base > addr + len)
8376                                 break;
8377                         if (seg->s_ops == &segdev_ops)
8378                                 continue;
8379                         if (((seg->s_ops != &segvn_ops) &&
8380                             (seg->s_ops != &segspt_shmops)) ||
8381                             ((SEGOP_GETVP(seg, addr, &vp) == 0 &&
8382                             vp != NULL && vp->v_type == VREG) &&
8383                             (SEGOP_GETTYPE(seg, addr) & MAP_SHARED))) {
8384                                 as_pageunlock(as, p->pparray,
8385                                     addr, len, p->s_flags);
8386                                 AS_LOCK_EXIT(as);
8386                                 AS_LOCK_EXIT(as, &as->a_lock);
8387                                 umem_decr_devlockmem(p);
8388                                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8389                                 *cookie = (ddi_umem_cookie_t)NULL;
8390                                 return (EFAULT);
8391                         }
8392                 }
8393                 AS_LOCK_EXIT(as);
8393                 AS_LOCK_EXIT(as, &as->a_lock);
8394         }


8397         /* Initialize the fields in the ddi_umem_cookie */
8398         p->cvaddr = addr;
8399         p->type = UMEM_LOCKED;
8400         if (driver_callback != NULL) {
8401                 /* i_ddi_umem_unlock and umem_lock_undo may need the cookie */
8402                 p->cook_refcnt = 2;
8403                 p->callbacks = *ops_vector;
8404         } else {
8405                 /* only i_ddi_umme_unlock needs the cookie */
8406                 p->cook_refcnt = 1;
```



```
8407         }

8409         *cookie = (ddi_umem_cookie_t)p;

8411         /*
8412          * If a driver callback was specified, add an entry to the
8413          * as struct callback list. The as_pagelock above guarantees
8414          * the persistence of as.
8415          */
8416         if (driver_callback) {
8417                 error = as_add_callback(as, umem_lock_undo, p, AS_ALL_EVENT,
8418                     addr, len, KM_SLEEP);
8419                 if (error != 0) {
8420                         as_pageunlock(as, p->pparray,
8421                             addr, len, p->s_flags);
8422                         umem_decr_devlockmem(p);
8423                         kmem_free(p, sizeof (struct ddi_umem_cookie));
8424                         *cookie = (ddi_umem_cookie_t)NULL;
8425                 }
8426         }
8427         return (error);
8428 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8504 Wed Nov 25 13:59:38 2015
new/usr/src/uts/common/os/urw.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 178 /*
 179  * Perform I/O to a given process. This will return EIO if we detect
 180  * corrupt memory and ENXIO if there is no such mapped address in the
 181  * user process's address space.
 182  */
 183 static int
 184 urw(proc_t *p, int writing, void *buf, size_t len, uintptr_t a)
 185 {
 186         caddr_t addr = (caddr_t)a;
 187         caddr_t page;
 188         caddr_t vaddr;
 189         struct seg *seg;
 190         int error = 0;
 191         int err = 0;
 192         uint_t prot;
 193         uint_t prot_rw = writing ? PROT_WRITE : PROT_READ;
 194         int protchanged;
 195         on_trap_data_t otd;
 196         int retrycnt;
 197         struct as *as = p->p_as;
 198         enum seg_rw rw;

 200         /*
 201          * Locate segment containing address of interest.
 202          */
 203         page = (caddr_t)(uintptr_t)((uintptr_t)addr & PAGEMASK);
 204         retrycnt = 0;
 205         AS_LOCK_ENTER(as, RW_WRITER);
 205         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 206 retry:
 207         if ((seg = as_segat(as, page)) == NULL ||
 208             !page_valid(seg, page)) {
 209                 AS_LOCK_EXIT(as);
 209                 AS_LOCK_EXIT(as, &as->a_lock);
 210                 return (ENXIO);
 211         }
 212         SEGOP_GETPROT(seg, page, 0, &prot);

 214         protchanged = 0;
 215         if ((prot & prot_rw) == 0) {
 216                 protchanged = 1;
 217                 err = SEGOP_SETPROT(seg, page, PAGESIZE, prot | prot_rw);

 219                 if (err == IE_RETRY) {
 220                         protchanged = 0;
 221                         ASSERT(retrycnt == 0);
 222                         retrycnt++;
 223                         goto retry;
 224                 }

 226                 if (err != 0) {
 227                         AS_LOCK_EXIT(as);
 227                         AS_LOCK_EXIT(as, &as->a_lock);
 228                         return (ENXIO);
 229                 }
 230         }

 232         /*
 233          * segvn may do a copy-on-write for F_SOFTLOCK/S_READ case to break
```



```
 234          * sharing to avoid a copy on write of a softlocked page by another
 235          * thread. But since we locked the address space as a writer no other
 236          * thread can cause a copy on write. S_READ_NOCOW is passed as the
 237          * access type to tell segvn that it's ok not to do a copy-on-write
 238          * for this SOFTLOCK fault.
 239          */
 240         if (writing)
 241                 rw = S_WRITE;
 242         else if (seg->s_ops == &segvn_ops)
 243                 rw = S_READ_NOCOW;
 244         else
 245                 rw = S_READ;

 247         if (SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
 248                 if (protchanged)
 249                         (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);
 250                 AS_LOCK_EXIT(as);
 250                 AS_LOCK_EXIT(as, &as->a_lock);
 251                 return (ENXIO);
 252         }
 253         CPU_STATS_ADD_K(vm, softlock, 1);

 255         /*
 256          * Make sure we're not trying to read or write off the end of the page.
 257          */
 258         ASSERT(len <= page + PAGESIZE - addr);

 260         /*
 261          * Map in the locked page, copy to our local buffer,
 262          * then map the page out and unlock it.
 263          */
 264         vaddr = mapin(as, addr, writing);

 266         /*
 267          * Since we are copying memory on behalf of the user process,
 268          * protect against memory error correction faults.
 269          */
 270         if (!on_trap(&otd, OT_DATA_EC)) {
 271                 if (seg->s_ops == &segdev_ops) {
 272                         /*
 273                          * Device memory can behave strangely; invoke
 274                          * a segdev-specific copy operation instead.
 275                          */
 276                         if (writing) {
 277                                 if (segdev_copyto(seg, addr, buf, vaddr, len))
 278                                         error = ENXIO;
 279                         } else {
 280                                 if (segdev_copyfrom(seg, addr, vaddr, buf, len))
 281                                         error = ENXIO;
 282                         }
 283                 } else {
 284                         if (writing)
 285                                 bcopy(buf, vaddr, len);
 286                         else
 287                                 bcopy(vaddr, buf, len);
 288                 }
 289         } else {
 290                 error = EIO;
 291         }
 292         no_trap();

 294         /*
 295          * If we're writing to an executable page, we may need to sychronize
 296          * the I$ with the modifications we made through the D$.
 297          */
 298         if (writing && (prot & PROT_EXEC))
```

```
 299                 sync_icache(vaddr, (uint_t)len);

 301         mapout(as, addr, vaddr, writing);

 303         if (rw == S_READ_NOCOW)
 304                 rw = S_READ;

 306         (void) SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTUNLOCK, rw);

 308         if (protchanged)
 309                 (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);

 311         AS_LOCK_EXIT(as);
 311         AS_LOCK_EXIT(as, &as->a_lock);

 313         return (error);
 314 }
```

_____unchanged_portion_omitted_

****
13956 Wed Nov 25 13:59:38 2015
new/usr/src/uts/common/os/vm_subr.c
patch as-lock-macro-simplification
****
_____***unchanged_portion_omitted_***

```
 318 #define MAX_MAPIN_PAGES 8

 320 /*
 321  * This function temporarily "borrows" user pages for kernel use. If
 322  * "cow" is on, it also sets up copy-on-write protection (only feasible
 323  * on MAP_PRIVATE segment) on the user mappings, to protect the borrowed
 324  * pages from any changes by the user. The caller is responsible for
 325  * unlocking and tearing down cow settings when it's done with the pages.
 326  * For an example, see kcfree().
 327  *
 328  * Pages behind [uaddr..uaddr+*lenp] under address space "as" are locked
 329  * (shared), and mapped into kernel address range [kaddr..kaddr+*lenp] if
 330  * kaddr != -1. On entering this function, cached_ppp contains a list
 331  * of pages that are mapped into [kaddr..kaddr+*lenp] already (from a
 332  * previous call). Thus if same pages remain behind [uaddr..uaddr+*lenp],
 333  * the kernel map won't need to be reloaded again.
 334  *
 335  * For cow == 1, if the pages are anonymous pages, it also bumps the anon
 336  * reference count, and change the user-mapping to read-only. This
 337  * scheme should work on all types of segment drivers. But to be safe,
 338  * we check against segvn here.
 339  *
 340  * Since this function is used to emulate copyin() semantic, it checks
 341  * to make sure the user-mappings allow "user-read".
 342  *
 343  * On exit "lenp" contains the number of bytes successfully locked and
 344  * mapped in. For the unsuccessful ones, the caller can fall back to
 345  * copyin().
 346  *
 347  * Error return:
 348  * ENOTSUP - operation like this is not supported either on this segment
 349  * type, or on this platform type.
 350  */
 351 int
 352 cow_mapin(struct as *as, caddr_t uaddr, caddr_t kaddr, struct page **cached_ppp,
 353     struct anon **app, size_t *lenp, int cow)
 354 {
 355         struct          hat *hat;
 356         struct seg      *seg;
 357         caddr_t         base;
 358         page_t          *pp, *ppp[MAX_MAPIN_PAGES];
 359         long            i;
 360         int             flags;
 361         size_t          size, total = *lenp;
 362         char            first = 1;
 363         faultcode_t     res;

 365         *lenp = 0;
 366         if (cow) {
 367                 AS_LOCK_ENTER(as, RW_WRITER);
 367                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 368                 seg = as_findseg(as, uaddr, 0);
 369                 if ((seg == NULL) || ((base = seg->s_base) > uaddr) ||
 370                     (uaddr + total) > base + seg->s_size) {
 371                         AS_LOCK_EXIT(as);
 371                         AS_LOCK_EXIT(as, &as->a_lock);
 372                         return (EINVAL);
 373                 }
 374                 /*
```



```
 375                  * The COW scheme should work for all segment types.
 376                  * But to be safe, we check against segvn.
 377                  */
 378                 if (seg->s_ops != &segvn_ops) {
 379                         AS_LOCK_EXIT(as);
 379                         AS_LOCK_EXIT(as, &as->a_lock);
 380                         return (ENOTSUP);
 381                 } else if ((SEGOP_GETTYPE(seg, uaddr) & MAP_PRIVATE) == 0) {
 382                         AS_LOCK_EXIT(as);
 382                         AS_LOCK_EXIT(as, &as->a_lock);
 383                         return (ENOTSUP);
 384                 }
 385         }
 386         hat = as->a_hat;
 387         size = total;
 388 tryagain:
 389         /*
 390          * If (cow), hat_softlock will also change the usr protection to RO.
 391          * This is the first step toward setting up cow. Before we
 392          * bump up an_refcnt, we can't allow any cow-fault on this
 393          * address. Otherwise segvn_fault will change the protection back
 394          * to RW upon seeing an_refcnt == 1.
 395          * The solution is to hold the writer lock on "as".
 396          */
 397         res = hat_softlock(hat, uaddr, &size, &ppp[0], cow ? HAT_COW : 0);
 398         size = total - size;
 399         *lenp += size;
 400         size = size >> PAGESHIFT;
 401         i = 0;
 402         while (i < size) {
 403                 pp = ppp[i];
 404                 if (cow) {
 405                         kmutex_t *ahm;
 406                         /*
 407                          * Another solution is to hold SE_EXCL on pp, and
 408                          * disable PROT_WRITE. This also works for MAP_SHARED
 409                          * segment. The disadvantage is that it locks the
 410                          * page from being used by anybody else.
 411                          */
 412                         ahm = AH_MUTEX(pp->p_vnode, pp->p_offset);
 413                         mutex_enter(ahm);
 414                         *app = swap_anon(pp->p_vnode, pp->p_offset);
 415                         /*
 416                          * Since we are holding the as lock, this avoids a
 417                          * potential race with anon_decref. (segvn_unmap and
 418                          * segvn_free needs the as writer lock to do anon_free.)
 419                          */
 420                         if (*app != NULL) {
 421 #if 0
 422                                 if ((*app)->an_refcnt == 0)
 423                                 /*
 424                                  * Consider the following senario (unlikey
 425                                  * though):
 426                                  * 1. an_refcnt == 2
 427                                  * 2. we solftlock the page.
 428                                  * 3. cow ocurrs on this addr. So a new ap,
 429                                  * page and mapping is established on addr.
 430                                  * 4. an_refcnt drops to 1 (segvn_faultpage
 431                                  * -> anon_decref(oldap))
 432                                  * 5. the last ref to ap also drops (from
 433                                  * another as). It ends up blocked inside
 434                                  * anon_decref trying to get page's excl lock.
 435                                  * 6. Later kcfree unlocks the page, call
 436                                  * anon_decref -> oops, ap is gone already.
 437                                  *
 438                                  * Holding as writer lock solves all problems.
```

```
439                                  */
440                                         *app = NULL;
441                                 else
442 #endif
443                                         (*app)->an_refcnt++;
444                                 }
445                                 mutex_exit(ahm);
446                         } else {
447                                 *app = NULL;
448                         }
449                         if (kaddr != (caddr_t)-1) {
450                                 if (pp != *cached_ppp) {
451                                         if (*cached_ppp == NULL)
452                                                 flags = HAT_LOAD_LOCK | HAT_NOSYNC |
453                                                     HAT_LOAD_NOCONSIST;
454                                         else
455                                                 flags = HAT_LOAD_REMAP |
456                                                     HAT_LOAD_NOCONSIST;
457                                         /*
458                                          * In order to cache the kernel mapping after
459                                          * the user page is unlocked, we call
460                                          * hat_devload instead of hat_memload so
461                                          * that the kernel mapping we set up here is
462                                          * "invisible" to the rest of the world. This
463                                          * is not very pretty. But as long as the
464                                          * caller bears the responsibility of keeping
465                                          * cache consistency, we should be ok -
466                                          * HAT_NOCONSIST will get us a uncached
467                                          * mapping on VAC. hat_softlock will flush
468                                          * a VAC_WRITEBACK cache. Therefore the kaddr
469                                          * doesn't have to be of the same vcolor as
470                                          * uaddr.
471                                          * The alternative is - change hat_devload
472                                          * to get a cached mapping. Allocate a kaddr
473                                          * with the same vcolor as uaddr. Then
474                                          * hat_softlock won't need to flush the VAC.
475                                          */
476                                         hat_devload(kas.a_hat, kaddr, PAGESIZE,
477                                             page_pptonum(pp), PROT_READ, flags);
478                                         *cached_ppp = pp;
479                                 }
480                                 kaddr += PAGESIZE;
481                         }
482                         cached_ppp++;
483                         app++;
484                         ++i;
485                 }
486                 if (cow) {
487                         AS_LOCK_EXIT(as);
488                         AS_LOCK_EXIT(as, &as->a_lock);
489                 }
490                 if (first && res == FC_NOMAP) {
491                         /*
492                          * If the address is not mapped yet, we call as_fault to
493                          * fault the pages in. We could've fallen back to copy and
494                          * let it fault in the pages. But for a mapped file, we
495                          * normally reference each page only once. For zero-copy to
496                          * be of any use, we'd better fall in the page now and try
497                          * again.
498                          */
499                         first = 0;
500                         size = size << PAGESHIFT;
501                         uaddr += size;
502                         total -= size;
503                         size = total;
```



```
504                         res = as_fault(as->a_hat, as, uaddr, size, F_INVAL, S_READ);
505                         if (cow)
506                                 AS_LOCK_ENTER(as, RW_WRITER);
507                                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
508                         goto tryagain;
509                 }
510                 switch (res) {
511                 case FC_NOSUPPORT:
512                         return (ENOTSUP);
513                 case FC_PROT:   /* Pretend we don't know about it. This will be */
514                                 /* caught by the caller when uiomove fails. */
515                 case FC_NOMAP:
516                 case FC_OBJERR:
517                 default:
518                         return (0);
                }
        }
_____unchanged_portion_omitted_
```

```
**********************************************************
   38724 Wed Nov 25 13:59:38 2015
new/usr/src/uts/common/os/watchpoint.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 153 #define X       0
 154 #define W       1
 155 #define R       2
 156 #define sum(a)  (a[X] + a[W] + a[R])

 158 /*
 159  * Common code for pr_mappage() and pr_unmappage().
 160  */
 161 static int
 162 pr_do_mappage(caddr_t addr, size_t size, int mapin, enum seg_rw rw, int kernel)
 163 {
 164         proc_t *p = curproc;
 165         struct as *as = p->p_as;
 166         char *eaddr = addr + size;
 167         int prot_rw = rw_to_prot(rw);
 168         int xrw = rw_to_index(rw);
 169         int rv = 0;
 170         struct watched_page *pwp;
 171         struct watched_page tpw;
 172         avl_index_t where;
 173         uint_t prot;

 175         ASSERT(as != &kas);

 177 startover:
 178         ASSERT(rv == 0);
 179         if (avl_numnodes(&as->a_wpage) == 0)
 180                 return (0);

 182         /*
 183          * as->a_wpage can only be changed while the process is totally stopped.
 184          * Don't grab p_lock here.  Holding p_lock while grabbing the address
 185          * space lock leads to deadlocks with the clock thread.  Note that if an
 186          * as_fault() is servicing a fault to a watched page on behalf of an
 187          * XHAT provider, watchpoint will be temporarily cleared (and wp_prot
 188          * will be set to wp_oprot).  Since this is done while holding as writer
 189          * lock, we need to grab as lock (reader lock is good enough).
 190          *
 191          * p_maplock prevents simultaneous execution of this function.  Under
 192          * normal circumstances, holdwatch() will stop all other threads, so the
 193          * lock isn't really needed.  But there may be multiple threads within
 194          * stop() when SWATCHOK is set, so we need to handle multiple threads
 195          * at once.  See holdwatch() for the details of this dance.
 196          */

 198         mutex_enter(&p->p_maplock);
 199         AS_LOCK_ENTER(as, RW_READER);
 199         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

 201         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
 202         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
 203                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

 205         for (; pwp != NULL && pwp->wp_vaddr < eaddr;
 206                 pwp = AVL_NEXT(&as->a_wpage, pwp)) {

 208                 /*
 209                  * If the requested protection has not been
 210                  * removed, we need not remap this page.
```



```
 211                  */
 212                 prot = pwp->wp_prot;
 213                 if (kernel || (prot & PROT_USER))
 214                         if (prot & prot_rw)
 215                                 continue;
 216                 /*
 217                  * If the requested access does not exist in the page's
 218                  * original protections, we need not remap this page.
 219                  * If the page does not exist yet, we can't test it.
 220                  */
 221                 if ((prot = pwp->wp_oprot) != 0) {
 222                         if (!(kernel || (prot & PROT_USER)))
 223                                 continue;
 224                         if (!(prot & prot_rw))
 225                                 continue;
 226                 }

 228                 if (mapin) {
 229                         /*
 230                          * Before mapping the page in, ensure that
 231                          * all other lwps are held in the kernel.
 232                          */
 233                         if (p->p_mapcnt == 0) {
 234                                 /*
 235                                  * Release as lock while in holdwatch()
 236                                  * in case other threads need to grab it.
 237                                  */
 238                                 AS_LOCK_EXIT(as);
 238                                 AS_LOCK_EXIT(as, &as->a_lock);
 239                                 mutex_exit(&p->p_maplock);
 240                                 if (holdwatch() != 0) {
 241                                         /*
 242                                          * We stopped in holdwatch().
 243                                          * Start all over again because the
 244                                          * watched page list may have changed.
 245                                          */
 246                                         goto startover;
 247                                 }
 248                                 mutex_enter(&p->p_maplock);
 249                                 AS_LOCK_ENTER(as, RW_READER);
 249                                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 250                         }
 251                         p->p_mapcnt++;
 252                 }

 254                 addr = pwp->wp_vaddr;
 255                 rv++;

 257                 prot = pwp->wp_prot;
 258                 if (mapin) {
 259                         if (kernel)
 260                                 pwp->wp_kmap[xrw]++;
 261                         else
 262                                 pwp->wp_umap[xrw]++;
 263                         pwp->wp_flags |= WP_NOWATCH;
 264                         if (pwp->wp_kmap[X] + pwp->wp_umap[X])
 265                                 /* cannot have exec-only protection */
 266                                 prot |= PROT_READ|PROT_EXEC;
 267                         if (pwp->wp_kmap[R] + pwp->wp_umap[R])
 268                                 prot |= PROT_READ;
 269                         if (pwp->wp_kmap[W] + pwp->wp_umap[W])
 270                                 /* cannot have write-only protection */
 271                                 prot |= PROT_READ|PROT_WRITE;
 272 #if 0   /* damned broken mmu feature! */
 273                         if (sum(pwp->wp_umap) == 0)
 274                                 prot &= ~PROT_USER;
```

```
#endif
                } else {
                        ASSERT(pwp->wp_flags & WP_NOWATCH);
                        if (kernel) {
                                ASSERT(pwp->wp_kmap[xrw] != 0);
                                --pwp->wp_kmap[xrw];
                        } else {
                                ASSERT(pwp->wp_umap[xrw] != 0);
                                --pwp->wp_umap[xrw];
                        }
                        if (sum(pwp->wp_kmap) + sum(pwp->wp_umap) == 0)
                                pwp->wp_flags &= ~WP_NOWATCH;
                        else {
                                if (pwp->wp_kmap[X] + pwp->wp_umap[X])
                                        /* cannot have exec-only protection */
                                        prot |= PROT_READ|PROT_EXEC;
                                if (pwp->wp_kmap[R] + pwp->wp_umap[R])
                                        prot |= PROT_READ;
                                if (pwp->wp_kmap[W] + pwp->wp_umap[W])
                                        /* cannot have write-only protection */
                                        prot |= PROT_READ|PROT_WRITE;
#if 0   /* damned broken mmu feature! */
                                if (sum(pwp->wp_umap) == 0)
                                        prot &= ~PROT_USER;
#endif
                        }
                }


                if (pwp->wp_oprot != 0) {       /* if page exists */
                        struct seg *seg;
                        uint_t oprot;
                        int err, retrycnt = 0;

                        AS_LOCK_EXIT(as);
                        AS_LOCK_ENTER(as, RW_WRITER);
                        AS_LOCK_EXIT(as, &as->a_lock);
                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
                retry:
                        seg = as_segat(as, addr);
                        ASSERT(seg != NULL);
                        SEGOP_GETPROT(seg, addr, 0, &oprot);
                        if (prot != oprot) {
                                err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
                                if (err == IE_RETRY) {
                                        ASSERT(retrycnt == 0);
                                        retrycnt++;
                                        goto retry;
                                }
                        }
                        AS_LOCK_EXIT(as);
                        AS_LOCK_EXIT(as, &as->a_lock);
                } else
                        AS_LOCK_EXIT(as);
                        AS_LOCK_EXIT(as, &as->a_lock);

                /*
                 * When all pages are mapped back to their normal state,
                 * continue the other lwps.
                 */
                if (!mapin) {
                        ASSERT(p->p_mapcnt > 0);
                        p->p_mapcnt--;
                        if (p->p_mapcnt == 0) {
                                mutex_exit(&p->p_maplock);
                                mutex_enter(&p->p_lock);
```



```
                                continuelwps(p);
                                mutex_exit(&p->p_lock);
                                mutex_enter(&p->p_maplock);
                        }
                }

                AS_LOCK_ENTER(as, RW_READER);
                AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
        }

        AS_LOCK_EXIT(as);
        AS_LOCK_EXIT(as, &as->a_lock);
        mutex_exit(&p->p_maplock);

        return (rv);
}
_____unchanged_portion_omitted_

/*
 * Function called by an lwp after it resumes from stop().
 */
void
setallwatch(void)
{
        proc_t *p = curproc;
        struct as *as = curproc->p_as;
        struct watched_page *pwp, *next;
        struct seg *seg;
        caddr_t vaddr;
        uint_t prot;
        int err, retrycnt;

        if (p->p_wprot == NULL)
                return;

        ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));

        AS_LOCK_ENTER(as, RW_WRITER);
        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

        pwp = p->p_wprot;
        while (pwp != NULL) {

                vaddr = pwp->wp_vaddr;
                retrycnt = 0;
        retry:
                ASSERT(pwp->wp_flags & WP_SETPROT);
                if ((seg = as_segat(as, vaddr)) != NULL &&
                    !(pwp->wp_flags & WP_NOWATCH)) {
                        prot = pwp->wp_prot;
                        err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
                        if (err == IE_RETRY) {
                                ASSERT(retrycnt == 0);
                                retrycnt++;
                                goto retry;
                        }
                }

                next = pwp->wp_list;

                if (pwp->wp_read + pwp->wp_write + pwp->wp_exec == 0) {
                        /*
                         * No watched areas remain in this page.
                         * Free the watched_page structure.
                         */
                        avl_remove(&as->a_wpage, pwp);
```

```
425                         kmem_free(pwp, sizeof (struct watched_page));
426                 } else {
427                         pwp->wp_flags &= ~WP_SETPROT;
428                 }

430                 pwp = next;
431         }
432         p->p_wprot = NULL;

434         AS_LOCK_EXIT(as);
434         AS_LOCK_EXIT(as, &as->a_lock);
435 }
```

_____unchanged_portion_omitted_

```
494 /*
495  * trap() calls here to determine if a fault is in a watched page.
496  * We return nonzero if this is true and the load/store would fail.
497  */
498 int
499 pr_is_watchpage(caddr_t addr, enum seg_rw rw)
500 {
501         struct as *as = curproc->p_as;
502         int rv;

504         if ((as == &kas) || avl_numnodes(&as->a_wpage) == 0)
505                 return (0);

507         /* Grab the lock because of XHAT (see comment in pr_mappage()) */
508         AS_LOCK_ENTER(as, RW_READER);
508         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
509         rv = pr_is_watchpage_as(addr, rw, as);
510         AS_LOCK_EXIT(as);
510         AS_LOCK_EXIT(as, &as->a_lock);

512         return (rv);
513 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  193150 Wed Nov 25 13:59:38 2015
new/usr/src/uts/common/os/zone.c
patch fix-bad-code
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

5593 /*
5594  * Return zero if the process has at least one vnode mapped in to its
5595  * address space which shouldn't be allowed to change zones.
5596  *
5597  * Also return zero if the process has any shared mappings which reserve
5598  * swap.  This is because the counting for zone.max-swap does not allow swap
5599  * reservation to be shared between zones.  zone swap reservation is counted
5600  * on zone->zone_max_swap.
5601  */
5602 static int
5603 as_can_change_zones(void)
5604 {
5605         proc_t *pp = curproc;
5606         struct seg *seg;
5607         struct as *as = pp->p_as;
5608         vnode_t *vp;
5609         int allow = 1;

5611         ASSERT(pp->p_as != &kas);
5612         AS_LOCK_ENTER(as, RW_READER);
5612         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
5613         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {

5615                 /*
5616                  * Cannot enter zone with shared anon memory which
5617                  * reserves swap.  See comment above.
5618                  */
5619                 if (seg_can_change_zones(seg) == B_FALSE) {
5620                         allow = 0;
5621                         break;
5622                 }
5623                 /*
5624                  * if we can't get a backing vnode for this segment then skip
5625                  * it.
5626                  */
5627                 vp = NULL;
5628                 if (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL)
5629                         continue;
5630                 if (!vn_can_change_zones(vp)) { /* bail on first match */
5631                         allow = 0;
5632                         break;
5633                 }
5634         }
5635         AS_LOCK_EXIT(as);
5635         AS_LOCK_EXIT(as, &as->a_lock);
5636         return (allow);
5637 }

5639 /*
5640  * Count swap reserved by curproc's address space
5641  */
5642 static size_t
5643 as_swresv(void)
5644 {
5645         proc_t *pp = curproc;
5646         struct seg *seg;
5647         struct as *as = pp->p_as;
5648         size_t swap = 0;
```



```
5650         ASSERT(pp->p_as != &kas);
5651         ASSERT(AS_WRITE_HELD(as));
5651         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
5652         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg))
5653                 swap += seg_swresv(seg);

5655         return (swap);
5656 }

5658 /*
5659  * Systemcall entry point for zone_enter().
5660  *
5661  * The current process is injected into said zone.  In the process
5662  * it will change its project membership, privileges, rootdir/cwd,
5663  * zone-wide rctls, and pool association to match those of the zone.
5664  *
5665  * The first zone_enter() called while the zone is in the ZONE_IS_READY
5666  * state will transition it to ZONE_IS_RUNNING.  Processes may only
5667  * enter a zone that is "ready" or "running".
5668  */
5669 static int
5670 zone_enter(zoneid_t zoneid)
5671 {
5672         zone_t *zone;
5673         vnode_t *vp;
5674         proc_t *pp = curproc;
5675         contract_t *ct;
5676         cont_process_t *ctp;
5677         task_t *tk, *oldtk;
5678         kproject_t *zone_proj0;
5679         cred_t *cr, *newcr;
5680         pool_t *oldpool, *newpool;
5681         sess_t *sp;
5682         uid_t uid;
5683         zone_status_t status;
5684         int err = 0;
5685         rctl_entity_p_t e;
5686         size_t swap;
5687         kthread_id_t t;

5689         if (secpolicy_zone_config(CRED()) != 0)
5690                 return (set_errno(EPERM));
5691         if (zoneid < MIN_USERZONEID || zoneid > MAX_ZONEID)
5692                 return (set_errno(EINVAL));

5694         /*
5695          * Stop all lwps so we don't need to hold a lock to look at
5696          * curproc->p_zone.  This needs to happen before we grab any
5697          * locks to avoid deadlock (another lwp in the process could
5698          * be waiting for the held lock).
5699          */
5700         if (curthread != pp->p_agenttp && !holdlwps(SHOLDFORK))
5701                 return (set_errno(EINTR));

5703         /*
5704          * Make sure we're not changing zones with files open or mapped in
5705          * to our address space which shouldn't be changing zones.
5706          */
5707         if (!files_can_change_zones()) {
5708                 err = EBADF;
5709                 goto out;
5710         }
5711         if (!as_can_change_zones()) {
5712                 err = EFAULT;
5713                 goto out;
```

```
5714         }

5716         mutex_enter(&zonehash_lock);
5717         if (pp->p_zone != global_zone) {
5718                 mutex_exit(&zonehash_lock);
5719                 err = EINVAL;
5720                 goto out;
5721         }

5723         zone = zone_find_all_by_id(zoneid);
5724         if (zone == NULL) {
5725                 mutex_exit(&zonehash_lock);
5726                 err = EINVAL;
5727                 goto out;
5728         }

5730         /*
5731          * To prevent processes in a zone from holding contracts on
5732          * extrazonal resources, and to avoid process contract
5733          * memberships which span zones, contract holders and processes
5734          * which aren't the sole members of their encapsulating process
5735          * contracts are not allowed to zone_enter.
5736          */
5737         ctp = pp->p_ct_process;
5738         ct = &ctp->conp_contract;
5739         mutex_enter(&ct->ct_lock);
5740         mutex_enter(&pp->p_lock);
5741         if ((avl_numnodes(&pp->p_ct_held) != 0) || (ctp->conp_nmembers != 1)) {
5742                 mutex_exit(&pp->p_lock);
5743                 mutex_exit(&ct->ct_lock);
5744                 mutex_exit(&zonehash_lock);
5745                 err = EINVAL;
5746                 goto out;
5747         }

5749         /*
5750          * Moreover, we don't allow processes whose encapsulating
5751          * process contracts have inherited extrazonal contracts.
5752          * While it would be easier to eliminate all process contracts
5753          * with inherited contracts, we need to be able to give a
5754          * restarted init (or other zone-penetrating process) its
5755          * predecessor's contracts.
5756          */
5757         if (ctp->conp_ninherited != 0) {
5758                 contract_t *next;
5759                 for (next = list_head(&ctp->conp_inherited); next;
5760                     next = list_next(&ctp->conp_inherited, next)) {
5761                         if (contract_getzuniqid(next) != zone->zone_uniqid) {
5762                                 mutex_exit(&pp->p_lock);
5763                                 mutex_exit(&ct->ct_lock);
5764                                 mutex_exit(&zonehash_lock);
5765                                 err = EINVAL;
5766                                 goto out;
5767                         }
5768                 }
5769         }

5771         mutex_exit(&pp->p_lock);
5772         mutex_exit(&ct->ct_lock);

5774         status = zone_status_get(zone);
5775         if (status < ZONE_IS_READY || status >= ZONE_IS_SHUTTING_DOWN) {
5776                 /*
5777                  * Can't join
5778                  */
5779                 mutex_exit(&zonehash_lock);
```



```
5780                 err = EINVAL;
5781                 goto out;
5782         }

5784         /*
5785          * Make sure new priv set is within the permitted set for caller
5786          */
5787         if (!priv_issubset(zone->zone_privset, &CR_OPPRIV(CRED()))) {
5788                 mutex_exit(&zonehash_lock);
5789                 err = EPERM;
5790                 goto out;
5791         }
5792         /*
5793          * We want to momentarily drop zonehash_lock while we optimistically
5794          * bind curproc to the pool it should be running in.  This is safe
5795          * since the zone can't disappear (we have a hold on it).
5796          */
5797         zone_hold(zone);
5798         mutex_exit(&zonehash_lock);

5800         /*
5801          * Grab pool_lock to keep the pools configuration from changing
5802          * and to stop ourselves from getting rebound to another pool
5803          * until we join the zone.
5804          */
5805         if (pool_lock_intr() != 0) {
5806                 zone_rele(zone);
5807                 err = EINTR;
5808                 goto out;
5809         }
5810         ASSERT(secpolicy_pool(CRED()) == 0);
5811         /*
5812          * Bind ourselves to the pool currently associated with the zone.
5813          */
5814         oldpool = curproc->p_pool;
5815         newpool = zone_pool_get(zone);
5816         if (pool_state == POOL_ENABLED && newpool != oldpool &&
5817             (err = pool_do_bind(newpool, P_PID, P_MYID,
5818             POOL_BIND_ALL)) != 0) {
5819                 pool_unlock();
5820                 zone_rele(zone);
5821                 goto out;
5822         }

5824         /*
5825          * Grab cpu_lock now; we'll need it later when we call
5826          * task_join().
5827          */
5828         mutex_enter(&cpu_lock);
5829         mutex_enter(&zonehash_lock);
5830         /*
5831          * Make sure the zone hasn't moved on since we dropped zonehash_lock.
5832          */
5833         if (zone_status_get(zone) >= ZONE_IS_SHUTTING_DOWN) {
5834                 /*
5835                  * Can't join anymore.
5836                  */
5837                 mutex_exit(&zonehash_lock);
5838                 mutex_exit(&cpu_lock);
5839                 if (pool_state == POOL_ENABLED &&
5840                     newpool != oldpool)
5841                         (void) pool_do_bind(oldpool, P_PID, P_MYID,
5842                             POOL_BIND_ALL);
5843                 pool_unlock();
5844                 zone_rele(zone);
5845                 err = EINVAL;
```

```
5846                    goto out;
5847            }

5849            /*
5850             * a_lock must be held while transfering locked memory and swap
5851             * reservation from the global zone to the non global zone because
5852             * asynchronous faults on the processes' address space can lock
5853             * memory and reserve swap via MCL_FUTURE and MAP_NORESERVE
5854             * segments respectively.
5855             */
5856            AS_LOCK_ENTER(pp->p_as, RW_WRITER);
5856            AS_LOCK_ENTER(pp->as, &pp->p_as->a_lock, RW_WRITER);
5857            swap = as_swresv();
5858            mutex_enter(&pp->p_lock);
5859            zone_proj0 = zone->zone_zsched->p_task->tk_proj;
5860            /* verify that we do not exceed and task or lwp limits */
5861            mutex_enter(&zone->zone_nlwps_lock);
5862            /* add new lwps to zone and zone's proj0 */
5863            zone_proj0->kpj_nlwps += pp->p_lwpcnt;
5864            zone->zone_nlwps += pp->p_lwpcnt;
5865            /* add 1 task to zone's proj0 */
5866            zone_proj0->kpj_ntasks += 1;

5868            zone_proj0->kpj_nprocs++;
5869            zone->zone_nprocs++;
5870            mutex_exit(&zone->zone_nlwps_lock);

5872            mutex_enter(&zone->zone_mem_lock);
5873            zone->zone_locked_mem += pp->p_locked_mem;
5874            zone_proj0->kpj_data.kpd_locked_mem += pp->p_locked_mem;
5875            zone->zone_max_swap += swap;
5876            mutex_exit(&zone->zone_mem_lock);

5878            mutex_enter(&(zone_proj0->kpj_data.kpd_crypto_lock));
5879            zone_proj0->kpj_data.kpd_crypto_mem += pp->p_crypto_mem;
5880            mutex_exit(&(zone_proj0->kpj_data.kpd_crypto_lock));

5882            /* remove lwps and process from proc's old zone and old project */
5883            mutex_enter(&pp->p_zone->zone_nlwps_lock);
5884            pp->p_zone->zone_nlwps -= pp->p_lwpcnt;
5885            pp->p_task->tk_proj->kpj_nlwps -= pp->p_lwpcnt;
5886            pp->p_task->tk_proj->kpj_nprocs--;
5887            pp->p_zone->zone_nprocs--;
5888            mutex_exit(&pp->p_zone->zone_nlwps_lock);

5890            mutex_enter(&pp->p_zone->zone_mem_lock);
5891            pp->p_zone->zone_locked_mem -= pp->p_locked_mem;
5892            pp->p_task->tk_proj->kpj_data.kpd_locked_mem -= pp->p_locked_mem;
5893            pp->p_zone->zone_max_swap -= swap;
5894            mutex_exit(&pp->p_zone->zone_mem_lock);

5896            mutex_enter(&(pp->p_task->tk_proj->kpj_data.kpd_crypto_lock));
5897            pp->p_task->tk_proj->kpj_data.kpd_crypto_mem -= pp->p_crypto_mem;
5898            mutex_exit(&(pp->p_task->tk_proj->kpj_data.kpd_crypto_lock));

5900            pp->p_flag |= SZONETOP;
5901            pp->p_zone = zone;
5902            mutex_exit(&pp->p_lock);
5903            AS_LOCK_EXIT(pp->p_as);
5903            AS_LOCK_EXIT(pp->p_as, &pp->p_as->a_lock);

5905            /*
5906             * Joining the zone cannot fail from now on.
5907             *
5908             * This means that a lot of the following code can be commonized and
5909             * shared with zsched().
```



```
5910             */

5912            /*
5913             * If the process contract fmri was inherited, we need to
5914             * flag this so that any contract status will not leak
5915             * extra zone information, svc_fmri in this case
5916             */
5917            if (ctp->conp_svc_ctid != ct->ct_id) {
5918                    mutex_enter(&ct->ct_lock);
5919                    ctp->conp_svc_zone_enter = ct->ct_id;
5920                    mutex_exit(&ct->ct_lock);
5921            }

5923            /*
5924             * Reset the encapsulating process contract's zone.
5925             */
5926            ASSERT(ct->ct_mzuniqid == GLOBAL_ZONEUNIQID);
5927            contract_setzuniqid(ct, zone->zone_uniqid);

5929            /*
5930             * Create a new task and associate the process with the project keyed
5931             * by (projid,zoneid).
5932             *
5933             * We might as well be in project 0; the global zone's projid doesn't
5934             * make much sense in a zone anyhow.
5935             *
5936             * This also increments zone_ntasks, and returns with p_lock held.
5937             */
5938            tk = task_create(0, zone);
5939            oldtk = task_join(tk, 0);
5940            mutex_exit(&cpu_lock);

5942            /*
5943             * call RCTLOP_SET functions on this proc
5944             */
5945            e.rcep_p.zone = zone;
5946            e.rcep_t = RCENTITY_ZONE;
5947            (void) rctl_set_dup(NULL, NULL, pp, &e, zone->zone_rctls, NULL,
5948                RCD_CALLBACK);
5949            mutex_exit(&pp->p_lock);

5951            /*
5952             * We don't need to hold any of zsched's locks here; not only do we know
5953             * the process and zone aren't going away, we know its session isn't
5954             * changing either.
5955             *
5956             * By joining zsched's session here, we mimic the behavior in the
5957             * global zone of init's sid being the pid of sched.  We extend this
5958             * to all zlogin-like zone_enter()'ing processes as well.
5959             */
5960            mutex_enter(&pidlock);
5961            sp = zone->zone_zsched->p_sessp;
5962            sess_hold(zone->zone_zsched);
5963            mutex_enter(&pp->p_lock);
5964            pgexit(pp);
5965            sess_rele(pp->p_sessp, B_TRUE);
5966            pp->p_sessp = sp;
5967            pgjoin(pp, zone->zone_zsched->p_pidp);

5969            /*
5970             * If any threads are scheduled to be placed on zone wait queue they
5971             * should abandon the idea since the wait queue is changing.
5972             * We need to be holding pidlock & p_lock to do this.
5973             */
5974            if ((t = pp->p_tlist) != NULL) {
5975                    do {
```

```
5976                        thread_lock(t);
5977                        /*
5978                         * Kick this thread so that he doesn't sit
5979                         * on a wrong wait queue.
5980                         */
5981                        if (ISWAITING(t))
5982                                setrun_locked(t);
5983
5984                        if (t->t_schedflag & TS_ANYWAITQ)
5985                                t->t_schedflag &= ~ TS_ANYWAITQ;
5986
5987                        thread_unlock(t);
5988                } while ((t = t->t_forw) != pp->p_tlist);
5989        }
5990
5991        /*
5992         * If there is a default scheduling class for the zone and it is not
5993         * the class we are currently in, change all of the threads in the
5994         * process to the new class.  We need to be holding pidlock & p_lock
5995         * when we call parmsset so this is a good place to do it.
5996         */
5997        if (zone->zone_defaultcid > 0 &&
5998            zone->zone_defaultcid != curthread->t_cid) {
5999                pcparms_t pcparms;
6000
6001                pcparms.pc_cid = zone->zone_defaultcid;
6002                pcparms.pc_clparms[0] = 0;
6003
6004                /*
6005                 * If setting the class fails, we still want to enter the zone.
6006                 */
6007                if ((t = pp->p_tlist) != NULL) {
6008                        do {
6009                                (void) parmsset(&pcparms, t);
6010                        } while ((t = t->t_forw) != pp->p_tlist);
6011                }
6012        }
6013
6014        mutex_exit(&pp->p_lock);
6015        mutex_exit(&pidlock);
6016
6017        mutex_exit(&zonehash_lock);
6018        /*
6019         * We're firmly in the zone; let pools progress.
6020         */
6021        pool_unlock();
6022        task_rele(oldtk);
6023        /*
6024         * We don't need to retain a hold on the zone since we already
6025         * incremented zone_ntasks, so the zone isn't going anywhere.
6026         */
6027        zone_rele(zone);
6028
6029        /*
6030         * Chroot
6031         */
6032        vp = zone->zone_rootvp;
6033        zone_chdir(vp, &PTOU(pp)->u_cdir, pp);
6034        zone_chdir(vp, &PTOU(pp)->u_rdir, pp);
6035
6036        /*
6037         * Change process credentials
6038         */
6039        newcr = cralloc();
6040        mutex_enter(&pp->p_crlock);
6041        cr = pp->p_cred;
```



```
6042        crcopy_to(cr, newcr);
6043        crsetzone(newcr, zone);
6044        pp->p_cred = newcr;
6045
6046        /*
6047         * Restrict all process privilege sets to zone limit
6048         */
6049        priv_intersect(zone->zone_privset, &CR_PPRIV(newcr));
6050        priv_intersect(zone->zone_privset, &CR_EPRIV(newcr));
6051        priv_intersect(zone->zone_privset, &CR_IPRIV(newcr));
6052        priv_intersect(zone->zone_privset, &CR_LPRIV(newcr));
6053        mutex_exit(&pp->p_crlock);
6054        crset(pp, newcr);
6055
6056        /*
6057         * Adjust upcount to reflect zone entry.
6058         */
6059        uid = crgetruid(newcr);
6060        mutex_enter(&pidlock);
6061        upcount_dec(uid, GLOBAL_ZONEID);
6062        upcount_inc(uid, zoneid);
6063        mutex_exit(&pidlock);
6064
6065        /*
6066         * Set up core file path and content.
6067         */
6068        set_core_defaults();
6069
6070 out:
6071        /*
6072         * Let the other lwps continue.
6073         */
6074        mutex_enter(&pp->p_lock);
6075        if (curthread != pp->p_agenttp)
6076                continuelwps(pp);
6077        mutex_exit(&pp->p_lock);
6078
6079        return (err != 0 ? set_errno(err) : 0);
6080 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52650 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/syscall/lgrpsys.c
patch as-lock-macro-simplification
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright 2015 Joyent, Inc.
 */

/*
 * lgroup system calls
 */

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/cpupart.h>
#include <sys/lgrp.h>
#include <sys/lgrp_user.h>
#include <sys/promif.h>         /* for prom_printf() */
#include <sys/sysmacros.h>
#include <sys/policy.h>

#include <vm/as.h>


/* definitions for mi_validity */
#define VALID_ADDR      1
#define VALID_REQ       2

/*
 * run through the given number of addresses and requests and return the
 * corresponding memory information for each address
 */
static int
meminfo(int addr_count, struct meminfo *mip)
{
        size_t          in_size, out_size, req_size, val_size;
        struct as       *as;
        struct hat      *hat;
        int             i, j, out_idx, info_count;
```



```
        lgrp_t          *lgrp;
        pfn_t           pfn;
        ssize_t         pgsz;
        int             *req_array, *val_array;
        uint64_t        *in_array, *out_array;
        uint64_t        addr, paddr;
        uintptr_t       vaddr;
        int             ret = 0;
        struct meminfo minfo;
#if defined(_SYSCALL32_IMPL)
        struct meminfo32 minfo32;
#endif

        /*
         * Make sure that there is at least one address to translate and
         * limit how many virtual addresses the kernel can do per call
         */
        if (addr_count < 1)
                return (set_errno(EINVAL));
        else if (addr_count > MAX_MEMINFO_CNT)
                addr_count = MAX_MEMINFO_CNT;

        if (get_udatamodel() == DATAMODEL_NATIVE) {
                if (copyin(mip, &minfo, sizeof (struct meminfo)))
                        return (set_errno(EFAULT));
        }
#if defined(_SYSCALL32_IMPL)
        else {
                bzero(&minfo, sizeof (minfo));
                if (copyin(mip, &minfo32, sizeof (struct meminfo32)))
                        return (set_errno(EFAULT));
                minfo.mi_inaddr = (const uint64_t *)(uintptr_t)
                    minfo32.mi_inaddr;
                minfo.mi_info_req = (const uint_t *)(uintptr_t)
                    minfo32.mi_info_req;
                minfo.mi_info_count = minfo32.mi_info_count;
                minfo.mi_outdata = (uint64_t *)(uintptr_t)
                    minfo32.mi_outdata;
                minfo.mi_validity = (uint_t *)(uintptr_t)
                    minfo32.mi_validity;
        }
#endif
        /*
         * all the input parameters have been copied in:-
         * addr_count - number of input addresses
         * minfo.mi_inaddr - array of input addresses
         * minfo.mi_info_req - array of types of information requested
         * minfo.mi_info_count - no. of pieces of info requested for each addr
         * minfo.mi_outdata - array into which the results are placed
         * minfo.mi_validity -  array containing bitwise result codes; 0th bit
         *                      evaluates validity of corresponding input
         *                      address, 1st bit validity of response to first
         *                      member of info_req, etc.
         */

        /* make sure mi_info_count is within limit */
        info_count = minfo.mi_info_count;
        if (info_count < 1 || info_count > MAX_MEMINFO_REQ)
                return (set_errno(EINVAL));

        /*
         * allocate buffer in_array for the input addresses and copy them in
         */
        in_size = sizeof (uint64_t) * addr_count;
        in_array = kmem_alloc(in_size, KM_SLEEP);
        if (copyin(minfo.mi_inaddr, in_array, in_size)) {
```

```
128                 kmem_free(in_array, in_size);
129                 return (set_errno(EFAULT));
130         }

132         /*
133          * allocate buffer req_array for the input info_reqs and copy them in
134          */
135         req_size = sizeof (uint_t) * info_count;
136         req_array = kmem_alloc(req_size, KM_SLEEP);
137         if (copyin(minfo.mi_info_req, req_array, req_size)) {
138                 kmem_free(req_array, req_size);
139                 kmem_free(in_array, in_size);
140                 return (set_errno(EFAULT));
141         }

143         /*
144          * Validate privs for each req.
145          */
146         for (i = 0; i < info_count; i++) {
147                 switch (req_array[i] & MEMINFO_MASK) {
148                 case MEMINFO_VLGRP:
149                 case MEMINFO_VPAGESIZE:
150                         break;
151                 default:
152                         if (secpolicy_meminfo(CRED()) != 0) {
153                                 kmem_free(req_array, req_size);
154                                 kmem_free(in_array, in_size);
155                                 return (set_errno(EPERM));
156                         }
157                         break;
158                 }
159         }

161         /*
162          * allocate buffer out_array which holds the results and will have
163          * to be copied out later
164          */
165         out_size = sizeof (uint64_t) * addr_count * info_count;
166         out_array = kmem_alloc(out_size, KM_SLEEP);

168         /*
169          * allocate buffer val_array which holds the validity bits and will
170          * have to be copied out later
171          */
172         val_size = sizeof (uint_t) * addr_count;
173         val_array = kmem_alloc(val_size, KM_SLEEP);

175         if ((req_array[0] & MEMINFO_MASK) == MEMINFO_PLGRP) {
176                 /* find the corresponding lgroup for each physical address */
177                 for (i = 0; i < addr_count; i++) {
178                         paddr = in_array[i];
179                         pfn = btop(paddr);
180                         lgrp = lgrp_pfn_to_lgrp(pfn);
181                         if (lgrp) {
182                                 out_array[i] = lgrp->lgrp_id;
183                                 val_array[i] = VALID_ADDR | VALID_REQ;
184                         } else {
185                                 out_array[i] = NULL;
186                                 val_array[i] = 0;
187                         }
188                 }
189         } else {
190                 /* get the corresponding memory info for each virtual address */
191                 as = curproc->p_as;

193                 AS_LOCK_ENTER(as, RW_READER);
```



```
193                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
194                 hat = as->a_hat;
195                 for (i = out_idx = 0; i < addr_count; i++, out_idx +=
196                     info_count) {
197                         addr = in_array[i];
198                         vaddr = (uintptr_t)(addr & ~PAGEOFFSET);
199                         if (!as_segat(as, (caddr_t)vaddr)) {
200                                 val_array[i] = 0;
201                                 continue;
202                         }
203                         val_array[i] = VALID_ADDR;
204                         pfn = hat_getpfnum(hat, (caddr_t)vaddr);
205                         if (pfn != PFN_INVALID) {
206                                 paddr = (uint64_t)((pfn << PAGESHIFT) |
207                                     (addr & PAGEOFFSET));
208                                 for (j = 0; j < info_count; j++) {
209                                         switch (req_array[j] & MEMINFO_MASK) {
210                                         case MEMINFO_VPHYSICAL:
211                                                 /*
212                                                  * return the physical address
213                                                  * corresponding to the input
214                                                  * virtual address
215                                                  */
216                                                 out_array[out_idx + j] = paddr;
217                                                 val_array[i] |= VALID_REQ << j;
218                                                 break;
219                                         case MEMINFO_VLGRP:
220                                                 /*
221                                                  * return the lgroup of physical
222                                                  * page corresponding to the
223                                                  * input virtual address
224                                                  */
225                                                 lgrp = lgrp_pfn_to_lgrp(pfn);
226                                                 if (lgrp) {
227                                                         out_array[out_idx + j] =
228                                                             lgrp->lgrp_id;
229                                                         val_array[i] |=
230                                                             VALID_REQ << j;
231                                                 }
232                                                 break;
233                                         case MEMINFO_VPAGESIZE:
234                                                 /*
235                                                  * return the size of physical
236                                                  * page corresponding to the
237                                                  * input virtual address
238                                                  */
239                                                 pgsz = hat_getpagesize(hat,
240                                                     (caddr_t)vaddr);
241                                                 if (pgsz != -1) {
242                                                         out_array[out_idx + j] =
243                                                             pgsz;
244                                                         val_array[i] |=
245                                                             VALID_REQ << j;
246                                                 }
247                                                 break;
248                                         case MEMINFO_VREPLCNT:
249                                                 /*
250                                                  * for future use:-
251                                                  * return the no. replicated
252                                                  * physical pages corresponding
253                                                  * to the input virtual address,
254                                                  * so it is always 0 at the
255                                                  * moment
256                                                  */
257                                                 out_array[out_idx + j] = 0;
258                                                 val_array[i] |= VALID_REQ << j;
```

```
259                                             break;
260                                     case MEMINFO_VREPL:
261                                             /*
262                                              * for future use:-
263                                              * return the nth physical
264                                              * replica of the specified
265                                              * virtual address
266                                              */
267                                             break;
268                                     case MEMINFO_VREPL_LGRP:
269                                             /*
270                                              * for future use:-
271                                              * return the lgroup of nth
272                                              * physical replica of the
273                                              * specified virtual address
274                                              */
275                                             break;
276                                     case MEMINFO_PLGRP:
277                                             /*
278                                              * this is for physical address
279                                              * only, shouldn't mix with
280                                              * virtual address
281                                              */
282                                             break;
283                                     default:
284                                             break;
285                                     }
286                                 }
287                         }
288                 }
289                 AS_LOCK_EXIT(as);
289                 AS_LOCK_EXIT(as, &as->a_lock);
290         }

292         /* copy out the results and validity bits and free the buffers */
293         if ((copyout(out_array, minfo.mi_outdata, out_size) != 0) ||
294             (copyout(val_array, minfo.mi_validity, val_size) != 0))
295                 ret = set_errno(EFAULT);

297         kmem_free(in_array, in_size);
298         kmem_free(out_array, out_size);
299         kmem_free(req_array, req_size);
300         kmem_free(val_array, val_size);

302         return (ret);
303 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   12498 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/syscall/rlimit.c
patch as-lock-macro-simplification
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/param.h>
#include <sys/types.h>
#include <sys/inttypes.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/tuneable.h>
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/proc.h>
#include <sys/resource.h>
#include <sys/ulimit.h>
#include <sys/debug.h>
#include <sys/rctl.h>

#include <vm/as.h>

/*
 * Perhaps ulimit could be moved into a user library, as calls to
 * getrlimit and setrlimit, were it not for binary compatibility
 * restrictions.
 */
long
ulimit(int cmd, long arg)
{
        proc_t *p = curproc;
        long    retval;
```



```
        switch (cmd) {

        case UL_GFILLIM: /* Return current file size limit. */
        {
                rlim64_t filesize;

                mutex_enter(&p->p_lock);
                filesize = rctl_enforced_value(rctlproc_legacy[RLIMIT_FSIZE],
                    p->p_rctls, p);
                mutex_exit(&p->p_lock);

                if (get_udatamodel() == DATAMODEL_ILP32) {
                        /*
                         * File size is returned in blocks for ulimit.
                         * This function is deprecated and therefore LFS API
                         * didn't define the behaviour of ulimit.
                         * Here we return maximum value of file size possible
                         * so that applications that do not check errors
                         * continue to work.
                         */
                        if (filesize > MAXOFF32_T)
                                filesize = MAXOFF32_T;
                        retval = ((int)filesize >> SCTRSHFT);
                } else
                        retval = filesize >> SCTRSHFT;
                break;
        }

        case UL_SFILLIM: /* Set new file size limit. */
        {
                int error = 0;
                rlim64_t lim = (rlim64_t)arg;
                struct rlimit64 rl64;
                rctl_alloc_gp_t *gp = rctl_rlimit_set_prealloc(1);

                if (lim >= (((rlim64_t)MAXOFFSET_T) >> SCTRSHFT))
                        lim = (rlim64_t)RLIM64_INFINITY;
                else
                        lim <<= SCTRSHFT;

                rl64.rlim_max = rl64.rlim_cur = lim;
                mutex_enter(&p->p_lock);
                if (error = rctl_rlimit_set(rctlproc_legacy[RLIMIT_FSIZE], p,
                    &rl64, gp, RCTL_LOCAL_DENY | RCTL_LOCAL_SIGNAL, SIGXFSZ,
                    CRED())) {
                        mutex_exit(&p->p_lock);
                        rctl_prealloc_destroy(gp);
                        return (set_errno(error));
                }
                mutex_exit(&p->p_lock);
                rctl_prealloc_destroy(gp);
                retval = arg;
                break;
        }

        case UL_GMEMLIM: /* Return maximum possible break value. */
        {
                struct seg *seg;
                struct seg *nextseg;
                struct as *as = p->p_as;
                caddr_t brkend;
                caddr_t brkbase;
                size_t size;
                rlim64_t size_ctl;
                rlim64_t vmem_ctl;
```

```
128                 /*
129                  * Find the segment with a virtual address
130                  * greater than the end of the current break.
131                  */
132                 nextseg = NULL;
133                 mutex_enter(&p->p_lock);
134                 brkbase = (caddr_t)p->p_brkbase;
135                 brkend = (caddr_t)p->p_brkbase + p->p_brksize;
136                 mutex_exit(&p->p_lock);

138                 /*
139                  * Since we can't return less than the current break,
140                  * initialize the return value to the current break
141                  */
142                 retval = (long)brkend;

144                 AS_LOCK_ENTER(as, RW_READER);
144                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
145                 for (seg = as_findseg(as, brkend, 0); seg != NULL;
146                     seg = AS_SEGNEXT(as, seg)) {
147                         if (seg->s_base >= brkend) {
148                                 nextseg = seg;
149                                 break;
150                         }
151                 }

153                 mutex_enter(&p->p_lock);
154                 size_ctl = rctl_enforced_value(rctlproc_legacy[RLIMIT_DATA],
155                     p->p_rctls, p);
156                 vmem_ctl = rctl_enforced_value(rctlproc_legacy[RLIMIT_VMEM],
157                     p->p_rctls, p);
158                 mutex_exit(&p->p_lock);

160                 /*
161                  * First, calculate the maximum break value based on
162                  * the user's RLIMIT_DATA, but also taking into account
163                  * that this value cannot be greater than as->a_userlimit.
164                  * We also take care to make sure that we don't overflow
165                  * in the calculation.
166                  */
167                 /*
168                  * Since we are casting the RLIMIT_DATA value to a
169                  * ulong (a 32-bit value in the 32-bit kernel) we have
170                  * to pass this assertion.
171                  */
172                 ASSERT32((size_t)size_ctl <= UINT32_MAX);

174                 size = (size_t)size_ctl;
175                 if (as->a_userlimit - brkbase > size)
176                         retval = MAX((size_t)retval, (size_t)(brkbase + size));
177                                         /* don't return less than current */
178                 else
179                         retval = (long)as->a_userlimit;

181                 /*
182                  * The max break cannot extend into the next segment
183                  */
184                 if (nextseg != NULL)
185                         retval = MIN((uintptr_t)retval,
186                             (uintptr_t)nextseg->s_base);

188                 /*
189                  * Handle the case where there is an limit on RLIMIT_VMEM
190                  */
191                 if (vmem_ctl < UINT64_MAX) {
192                         /* calculate brkend based on the end of page */
```



```
193                         caddr_t brkendpg = (caddr_t)roundup((uintptr_t)brkend,
194                             PAGESIZE);
195                         /*
196                          * Large Files: The following assertion has to pass
197                          * through to ensure the correctness of the cast.
198                          */
199                         ASSERT32(vmem_ctl <= UINT32_MAX);

201                         size = (size_t)(vmem_ctl & PAGEMASK);

203                         if (as->a_size < size)
204                                 size -= as->a_size;
205                         else
206                                 size = 0;
207                         /*
208                          * Take care to not overflow the calculation
209                          */
210                         if (as->a_userlimit - brkendpg > size)
211                                 retval = MIN((size_t)retval,
212                                     (size_t)(brkendpg + size));
213                 }

215                 AS_LOCK_EXIT(as);
215                 AS_LOCK_EXIT(as, &as->a_lock);

217                 /* truncate to same boundary as sbrk */

219                 switch (get_udatamodel()) {
220                 default:
221                 case DATAMODEL_ILP32:
222                         retval = retval & ~(8-1);
223                         break;
224                 case DATAMODEL_LP64:
225                         retval = retval & ~(16-1);
226                         break;
227                 }
228                 break;
229         }

231         case UL_GDESLIM: /* Return approximate number of open files */
232         {
233                 rlim64_t fdno_ctl;

235                 mutex_enter(&curproc->p_lock);
236                 fdno_ctl = rctl_enforced_value(rctlproc_legacy[RLIMIT_NOFILE],
237                     curproc->p_rctls, curproc);
238                 ASSERT(fdno_ctl <= INT_MAX);
239                 retval = (rlim_t)fdno_ctl;
240                 mutex_exit(&curproc->p_lock);
241                 break;
242         }

244         default:
245                 return (set_errno(EINVAL));

247         }
248         return (retval);
249 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   23723 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/syscall/utssys.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

313 static fu_data_t *
314 dofusers(vnode_t *fvp, int flags)
315 {
316         fu_data_t       *fu_data;
317         proc_t          *prp;
318         vfs_t           *cvfsp;
319         pid_t           npids, pidx, *pidlist;
320         int             v_proc = v.v_proc;      /* max # of procs */
321         int             pcnt = 0;
322         int             contained = (flags & F_CONTAINED);
323         int             nbmandonly = (flags & F_NBMANDLIST);
324         int             dip_usage = (flags & F_DEVINFO);
325         int             fvp_isdev = vn_matchops(fvp, spec_getvnodeops());
326         zone_t *zone = curproc->p_zone;
327         int inglobal = INGLOBALZONE(curproc);

329         /* get a pointer to the file system containing this vnode */
330         cvfsp = fvp->v_vfsp;
331         ASSERT(cvfsp);

333         /* allocate the data structure to return our results in */
334         fu_data = kmem_alloc(fu_data_size(v_proc), KM_SLEEP);
335         fu_data->fud_user_max = v_proc;
336         fu_data->fud_user_count = 0;

338         /* get a snapshot of all the pids we're going to check out */
339         pidlist = kmem_alloc(v_proc * sizeof (pid_t), KM_SLEEP);
340         mutex_enter(&pidlock);
341         for (npids = 0, prp = practive; prp != NULL; prp = prp->p_next) {
342                 if (inglobal || prp->p_zone == zone)
343                         pidlist[npids++] = prp->p_pid;
344         }
345         mutex_exit(&pidlock);

347         /* grab each process and check its file usage */
348         for (pidx = 0; pidx < npids; pidx++) {
349                 locklist_t      *llp = NULL;
350                 uf_info_t       *fip;
351                 vnode_t         *vp;
352                 user_t          *up;
353                 sess_t          *sp;
354                 uid_t           uid;
355                 pid_t           pid = pidlist[pidx];
356                 int             i, use_flag = 0;

358                 /*
359                  * grab prp->p_lock using sprlock()
360                  * if sprlock() fails the process does not exists anymore
361                  */
362                 prp = sprlock(pid);
363                 if (prp == NULL)
364                         continue;

366                 /* get the processes credential info in case we need it */
367                 mutex_enter(&prp->p_crlock);
368                 uid = crgetruid(prp->p_cred);
369                 mutex_exit(&prp->p_crlock);

371                 /*
```



```
372                  * it's safe to drop p_lock here because we
373                  * called sprlock() before and it set the SPRLOCK
374                  * flag for the process so it won't go away.
375                  */
376                 mutex_exit(&prp->p_lock);

378                 /*
379                  * now we want to walk a processes open file descriptors
380                  * to do this we need to grab the fip->fi_lock.  (you
381                  * can't hold p_lock when grabbing the fip->fi_lock.)
382                  */
383                 fip = P_FINFO(prp);
384                 mutex_enter(&fip->fi_lock);

386                 /*
387                  * Snapshot nbmand locks for pid
388                  */
389                 llp = flk_active_nbmand_locks(prp->p_pid);
390                 for (i = 0; i < fip->fi_nfiles; i++) {
391                         uf_entry_t      *ufp;
392                         file_t          *fp;

394                         UF_ENTER(ufp, fip, i);
395                         if (((fp = ufp->uf_file) == NULL) ||
396                             ((vp = fp->f_vnode) == NULL)) {
397                                 UF_EXIT(ufp);
398                                 continue;
399                         }

401                         /*
402                          * if the target file (fvp) is not a device
403                          * and corrosponds to the root of a filesystem
404                          * (cvfsp), then check if it contains the file
405                          * is use by this process (vp).
406                          */
407                         if (contained && (vp->v_vfsp == cvfsp))
408                                 use_flag |= F_OPEN;

410                         /*
411                          * if the target file (fvp) is not a device,
412                          * then check if it matches the file in use
413                          * by this process (vp).
414                          */
415                         if (!fvp_isdev && VN_CMP(fvp, vp))
416                                 use_flag |= F_OPEN;

418                         /*
419                          * if the target file (fvp) is a device,
420                          * then check if the current file in use
421                          * by this process (vp) maps to the same device
422                          * minor node.
423                          */
424                         if (fvp_isdev &&
425                             vn_matchops(vp, spec_getvnodeops()) &&
426                             (fvp->v_rdev == vp->v_rdev))
427                                 use_flag |= F_OPEN;

429                         /*
430                          * if the target file (fvp) is a device,
431                          * and we're checking for device instance
432                          * usage, then check if the current file in use
433                          * by this process (vp) maps to the same device
434                          * instance.
435                          */
436                         if (dip_usage &&
437                             vn_matchops(vp, spec_getvnodeops()) &&
```

```
438                             (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
439                                 use_flag |= F_OPEN;

441                         /*
442                          * if the current file in use by this process (vp)
443                          * doesn't match what we're looking for, move on
444                          * to the next file in the process.
445                          */
446                         if ((use_flag & F_OPEN) == 0) {
447                                 UF_EXIT(ufp);
448                                 continue;
449                         }

451                         if (proc_has_nbmand_on_vp(vp, prp->p_pid, llp)) {
452                                 /* A nbmand found so we're done.  */
453                                 use_flag |= F_NBM;
454                                 UF_EXIT(ufp);
455                                 break;
456                         }
457                         UF_EXIT(ufp);
458                 }
459                 if (llp)
460                         flk_free_locklist(llp);

462                 mutex_exit(&fip->fi_lock);

464                 /*
465                  * If nbmand usage tracking is desired and no nbmand was
466                  * found for this process, then no need to do further
467                  * usage tracking for this process.
468                  */
469                 if (nbmandonly && (!(use_flag & F_NBM))) {
470                         /*
471                          * grab the process lock again, clear the SPRLOCK
472                          * flag, release the process, and continue.
473                          */
474                         mutex_enter(&prp->p_lock);
475                         sprunlock(prp);
476                         continue;
477                 }

479                 /*
480                  * All other types of usage.
481                  * For the next few checks we need to hold p_lock.
482                  */
483                 mutex_enter(&prp->p_lock);
484                 up = PTOU(prp);
485                 if (fvp_isdev) {
486                         /*
487                          * if the target file (fvp) is a device
488                          * then check if it matches the processes tty
489                          *
490                          * we grab s_lock to protect ourselves against
491                          * freectty() freeing the vnode out from under us.
492                          */
493                         sp = prp->p_sessp;
494                         mutex_enter(&sp->s_lock);
495                         vp = prp->p_sessp->s_vp;
496                         if (vp != NULL) {
497                                 if (fvp->v_rdev == vp->v_rdev)
498                                         use_flag |= F_TTY;

500                                 if (dip_usage &&
501                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
502                                         use_flag |= F_TTY;
503                         }
```



```
504                         mutex_exit(&sp->s_lock);
505                 } else {
506                         /* check the processes current working directory */
507                         if (up->u_cdir &&
508                             (VN_CMP(fvp, up->u_cdir) ||
509                             (contained && (up->u_cdir->v_vfsp == cvfsp))))
510                                 use_flag |= F_CDIR;

512                         /* check the processes root directory */
513                         if (up->u_rdir &&
514                             (VN_CMP(fvp, up->u_rdir) ||
515                             (contained && (up->u_rdir->v_vfsp == cvfsp))))
516                                 use_flag |= F_RDIR;

518                         /* check the program text vnode */
519                         if (prp->p_exec &&
520                             (VN_CMP(fvp, prp->p_exec) ||
521                             (contained && (prp->p_exec->v_vfsp == cvfsp))))
522                                 use_flag |= F_TEXT;
523                 }

525                 /* Now we can drop p_lock again */
526                 mutex_exit(&prp->p_lock);

528                 /*
529                  * now we want to walk a processes memory mappings.
530                  * to do this we need to grab the prp->p_as lock.  (you
531                  * can't hold p_lock when grabbing the prp->p_as lock.)
532                  */
533                 if (prp->p_as != &kas) {
534                         struct seg      *seg;
535                         struct as       *as = prp->p_as;

537                         AS_LOCK_ENTER(as, RW_READER);
537                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
538                         for (seg = AS_SEGFIRST(as); seg;
539                             seg = AS_SEGNEXT(as, seg)) {
540                                 /*
541                                  * if we can't get a backing vnode for this
542                                  * segment then skip it
543                                  */
544                                 vp = NULL;
545                                 if ((SEGOP_GETVP(seg, seg->s_base, &vp)) ||
546                                     (vp == NULL))
547                                         continue;

549                                 /*
550                                  * if the target file (fvp) is not a device
551                                  * and corrosponds to the root of a filesystem
552                                  * (cvfsp), then check if it contains the
553                                  * vnode backing this segment (vp).
554                                  */
555                                 if (contained && (vp->v_vfsp == cvfsp)) {
556                                         use_flag |= F_MAP;
557                                         break;
558                                 }

560                                 /*
561                                  * if the target file (fvp) is not a device,
562                                  * check if it matches the the vnode backing
563                                  * this segment (vp).
564                                  */
565                                 if (!fvp_isdev && VN_CMP(fvp, vp)) {
566                                         use_flag |= F_MAP;
567                                         break;
568                                 }
```

```
570                                 /*
571                                  * if the target file (fvp) isn't a device,
572                                  * or the the vnode backing this segment (vp)
573                                  * isn't a device then continue.
574                                  */
575                                 if (!fvp_isdev ||
576                                     !vn_matchops(vp, spec_getvnodeops()))
577                                         continue;

579                                 /*
580                                  * check if the vnode backing this segment
581                                  * (vp) maps to the same device minor node
582                                  * as the target device (fvp)
583                                  */
584                                 if (fvp->v_rdev == vp->v_rdev) {
585                                         use_flag |= F_MAP;
586                                         break;
587                                 }

589                                 /*
590                                  * if we're checking for device instance
591                                  * usage, then check if the vnode backing
592                                  * this segment (vp) maps to the same device
593                                  * instance as the target device (fvp).
594                                  */
595                                 if (dip_usage &&
596                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip)) {
597                                         use_flag |= F_MAP;
598                                         break;
599                                 }
600                         }
601                         AS_LOCK_EXIT(as);
601                         AS_LOCK_EXIT(as, &as->a_lock);
602                 }

604                 if (use_flag) {
605                         ASSERT(pcnt < fu_data->fud_user_max);
606                         fu_data->fud_user[pcnt].fu_flags = use_flag;
607                         fu_data->fud_user[pcnt].fu_pid = pid;
608                         fu_data->fud_user[pcnt].fu_uid = uid;
609                         pcnt++;
610                 }

612                 /*
613                  * grab the process lock again, clear the SPRLOCK
614                  * flag, release the process, and continue.
615                  */
616                 mutex_enter(&prp->p_lock);
617                 sprunlock(prp);
618         }

620         kmem_free(pidlist, v_proc * sizeof (pid_t));

622         fu_data->fud_user_count = pcnt;
623         return (fu_data);
624 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11767 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/vm/as.h
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

#ifdef _KERNEL

/*
 * Flags for as_gap.
 */
#define AH_DIR          0x1     /* direction flag mask */
#define AH_LO           0x0     /* find lowest hole */
#define AH_HI           0x1     /* find highest hole */
#define AH_CONTAIN      0x2     /* hole must contain `addr' */

extern struct as kas;           /* kernel's address space */

/*
 * Macros for address space locking.  Note that we use RW_READER_STARVEWRITER
 * whenever we acquire the address space lock as reader to assure that it can
 * be used without regard to lock order in conjunction with filesystem locks.
 * This allows filesystems to safely induce user-level page faults with
 * filesystem locks held while concurrently allowing filesystem entry points
 * acquiring those same locks to be called with the address space lock held as
 * reader.  RW_READER_STARVEWRITER thus prevents reader/reader+RW_WRITE_WANTED
 * deadlocks in the style of fop_write()+as_fault()/as_*()+fop_putpage() and
 * fop_read()+as_fault()/as_*()+fop_getpage().  (See the Big Theory Statement
 * in rwlock.c for more information on the semantics of and motivation behind
 * RW_READER_STARVEWRITER.)
 */
#define AS_LOCK_ENTER(as, type)         rw_enter(&(as)->a_lock, \
#define AS_LOCK_ENTER(as, lock, type)           rw_enter((lock), \
        (type) == RW_READER ? RW_READER_STARVEWRITER : (type))
#define AS_LOCK_EXIT(as)                rw_exit(&(as)->a_lock)
#define AS_LOCK_DESTROY(as)             rw_destroy(&(as)->a_lock)
#define AS_LOCK_TRYENTER(as, type)      rw_tryenter(&(as)->a_lock, \
#define AS_LOCK_EXIT(as, lock)                  rw_exit((lock))
#define AS_LOCK_DESTROY(as, lock)               rw_destroy((lock))
#define AS_LOCK_TRYENTER(as, lock, type)        rw_tryenter((lock), \
        (type) == RW_READER ? RW_READER_STARVEWRITER : (type))

/*
 * Macros to test lock states.
 */
#define AS_LOCK_HELD(as)                RW_LOCK_HELD(&(as)->a_lock)
#define AS_READ_HELD(as)                RW_READ_HELD(&(as)->a_lock)
#define AS_WRITE_HELD(as)               RW_WRITE_HELD(&(as)->a_lock)
#define AS_LOCK_HELD(as, lock)          RW_LOCK_HELD((lock))
#define AS_READ_HELD(as, lock)          RW_READ_HELD((lock))
#define AS_WRITE_HELD(as, lock)         RW_WRITE_HELD((lock))

/*
 * macros to walk thru segment lists
 */
#define AS_SEGFIRST(as)         avl_first(&(as)->a_segtree)
#define AS_SEGNEXT(as, seg)     AVL_NEXT(&(as)->a_segtree, (seg))
#define AS_SEGPREV(as, seg)     AVL_PREV(&(as)->a_segtree, (seg))

void    as_init(void);
void    as_avlinit(struct as *);
struct  seg *as_segat(struct as *as, caddr_t addr);
void    as_rangelock(struct as *as);
void    as_rangeunlock(struct as *as);
struct  as *as_alloc();
```



```
void    as_free(struct as *as);
int     as_dup(struct as *as, struct proc *forkedproc);
struct  seg *as_findseg(struct as *as, caddr_t addr, int tail);
int     as_addseg(struct as *as, struct seg *newseg);
struct  seg *as_removeseg(struct as *as, struct seg *seg);
faultcode_t as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
                enum fault_type type, enum seg_rw rw);
faultcode_t as_faulta(struct as *as, caddr_t addr, size_t size);
int     as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
int     as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
int     as_unmap(struct as *as, caddr_t addr, size_t size);
int     as_map(struct as *as, caddr_t addr, size_t size, int ((*crfp)()),
                void *argsp);
void    as_purge(struct as *as);
int     as_gap(struct as *as, size_t minlen, caddr_t *basep, size_t *lenp,
                uint_t flags, caddr_t addr);
int     as_gap_aligned(struct as *as, size_t minlen, caddr_t *basep,
            size_t *lenp, uint_t flags, caddr_t addr, size_t align,
            size_t redzone, size_t off);

int     as_memory(struct as *as, caddr_t *basep, size_t *lenp);
size_t  as_swapout(struct as *as);
int     as_incore(struct as *as, caddr_t addr, size_t size, char *vec,
                size_t *sizep);
int     as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
                uintptr_t arg, ulong_t *lock_map, size_t pos);
int     as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
                size_t size, enum seg_rw rw);
void    as_pageunlock(struct as *as, struct page **pp, caddr_t addr,
                size_t size, enum seg_rw rw);
int     as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
                boolean_t wait);
int     as_set_default_lpsize(struct as *as, caddr_t addr, size_t size);
void    as_setwatch(struct as *as);
void    as_clearwatch(struct as *as);
int     as_getmemid(struct as *, caddr_t, memid_t *);

int     as_add_callback(struct as *, void (*)(), void *, uint_t,
                        caddr_t, size_t, int);
uint_t  as_delete_callback(struct as *, void *);

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
  113610 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/vm/seg_dev.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 356 /*
 357  * Create a device segment.
 358  */
 359 int
 360 segdev_create(struct seg *seg, void *argsp)
 361 {
 362         struct segdev_data *sdp;
 363         struct segdev_crargs *a = (struct segdev_crargs *)argsp;
 364         devmap_handle_t *dhp = (devmap_handle_t *)a->devmap_data;
 365         int error;

 367         /*
 368          * Since the address space is "write" locked, we
 369          * don't need the segment lock to protect "segdev" data.
 370          */
 371         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
 371         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 373         hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);

 375         sdp = sdp_alloc();

 377         sdp->mapfunc = a->mapfunc;
 378         sdp->offset = a->offset;
 379         sdp->prot = a->prot;
 380         sdp->maxprot = a->maxprot;
 381         sdp->type = a->type;
 382         sdp->pageprot = 0;
 383         sdp->softlockcnt = 0;
 384         sdp->vpage = NULL;

 386         if (sdp->mapfunc == NULL)
 387                 sdp->devmap_data = dhp;
 388         else
 389                 sdp->devmap_data = dhp = NULL;

 391         sdp->hat_flags = a->hat_flags;
 392         sdp->hat_attr = a->hat_attr;

 394         /*
 395          * Currently, hat_flags supports only HAT_LOAD_NOCONSIST
 396          */
 397         ASSERT(!(sdp->hat_flags & ~HAT_LOAD_NOCONSIST));

 399         /*
 400          * Hold shadow vnode -- segdev only deals with
 401          * character (VCHR) devices. We use the common
 402          * vp to hang pages on.
 403          */
 404         sdp->vp = specfind(a->dev, VCHR);
 405         ASSERT(sdp->vp != NULL);

 407         seg->s_ops = &segdev_ops;
 408         seg->s_data = sdp;

 410         while (dhp != NULL) {
 411                 dhp->dh_seg = seg;
 412                 dhp = dhp->dh_next;
 413         }
```



```
 415         /*
 416          * Inform the vnode of the new mapping.
 417          */
 418         /*
 419          * It is ok to use pass sdp->maxprot to ADDMAP rather than to use
 420          * dhp specific maxprot because spec_addmap does not use maxprot.
 421          */
 422         error = VOP_ADDMAP(VTOCVP(sdp->vp), sdp->offset,
 423             seg->s_as, seg->s_base, seg->s_size,
 424             sdp->prot, sdp->maxprot, sdp->type, CRED(), NULL);

 426         if (error != 0) {
 427                 sdp->devmap_data = NULL;
 428                 hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
 429                     HAT_UNLOAD_UNMAP);
 430         } else {
 431                 /*
 432                  * Mappings of /dev/null don't count towards the VSZ of a
 433                  * process.  Mappings of /dev/null have no mapping type.
 434                  */
 435                 if ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED |
 436                     MAP_PRIVATE)) == 0) {
 437                         seg->s_as->a_resvsize -= seg->s_size;
 438                 }
 439         }

 441         return (error);
 442 }
______unchanged_portion_omitted_

 455 /*
 456  * Duplicate seg and return new segment in newseg.
 457  */
 458 static int
 459 segdev_dup(struct seg *seg, struct seg *newseg)
 460 {
 461         struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
 462         struct segdev_data *newsdp;
 463         devmap_handle_t *dhp = (devmap_handle_t *)sdp->devmap_data;
 464         size_t npages;
 465         int ret;

 467         TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_DUP,
 468             "segdev_dup:start dhp=%p, seg=%p", (void *)dhp, (void *)seg);

 470         DEBUGF(3, (CE_CONT, "segdev_dup: dhp %p seg %p\n",
 471             (void *)dhp, (void *)seg));

 473         /*
 474          * Since the address space is "write" locked, we
 475          * don't need the segment lock to protect "segdev" data.
 476          */
 477         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
 477         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 479         newsdp = sdp_alloc();

 481         newseg->s_ops = seg->s_ops;
 482         newseg->s_data = (void *)newsdp;

 484         VN_HOLD(sdp->vp);
 485         newsdp->vp      = sdp->vp;
 486         newsdp->mapfunc = sdp->mapfunc;
 487         newsdp->offset  = sdp->offset;
 488         newsdp->pageprot = sdp->pageprot;
```

```
489         newsdp->prot    = sdp->prot;
490         newsdp->maxprot = sdp->maxprot;
491         newsdp->type = sdp->type;
492         newsdp->hat_attr = sdp->hat_attr;
493         newsdp->hat_flags = sdp->hat_flags;
494         newsdp->softlockcnt = 0;

496         /*
497          * Initialize per page data if the segment we are
498          * dup'ing has per page information.
499          */
500         npages = seg_pages(newseg);

502         if (sdp->vpage != NULL) {
503                 size_t nbytes = vpgtob(npages);

505                 newsdp->vpage = kmem_zalloc(nbytes, KM_SLEEP);
506                 bcopy(sdp->vpage, newsdp->vpage, nbytes);
507         } else
508                 newsdp->vpage = NULL;

510         /*
511          * duplicate devmap handles
512          */
513         if (dhp != NULL) {
514                 ret = devmap_handle_dup(dhp,
515                     (devmap_handle_t **)&newsdp->devmap_data, newseg);
516                 if (ret != 0) {
517                         TRACE_3(TR_FAC_DEVMAP, TR_DEVMAP_DUP_CK1,
518                             "segdev_dup:ret1 ret=%x, dhp=%p seg=%p",
519                             ret, (void *)dhp, (void *)seg);
520                         DEBUGF(1, (CE_CONT,
521                             "segdev_dup: ret %x dhp %p seg %p\n",
522                             ret, (void *)dhp, (void *)seg));
523                         return (ret);
524                 }
525         }

527         /*
528          * Inform the common vnode of the new mapping.
529          */
530         return (VOP_ADDMAP(VTOCVP(newsdp->vp),
531             newsdp->offset, newseg->s_as,
532             newseg->s_base, newseg->s_size, newsdp->prot,
533             newsdp->maxprot, sdp->type, CRED(), NULL));
534 }
```

*unchanged_portion_omitted_*

```
615 /*
616  * Split a segment at addr for length len.
617  */
618 /*ARGSUSED*/
619 static int
620 segdev_unmap(struct seg *seg, caddr_t addr, size_t len)
621 {
622         register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
623         register struct segdev_data *nsdp;
624         register struct seg *nseg;
625         register size_t opages;         /* old segment size in pages */
626         register size_t npages;         /* new segment size in pages */
627         register size_t dpages;         /* pages being deleted (unmapped) */
628         register size_t nbytes;
629         devmap_handle_t *dhp = (devmap_handle_t *)sdp->devmap_data;
630         devmap_handle_t *dhpp;
631         devmap_handle_t *newdhp;
632         struct devmap_callback_ctl *callbackops;
```



```
633         caddr_t nbase;
634         offset_t off;
635         ulong_t nsize;
636         size_t mlen, sz;

638         TRACE_4(TR_FAC_DEVMAP, TR_DEVMAP_UNMAP,
639             "segdev_unmap:start dhp=%p, seg=%p addr=%p len=%lx",
640             (void *)dhp, (void *)seg, (void *)addr, len);

642         DEBUGF(3, (CE_CONT, "segdev_unmap: dhp %p seg %p addr %p len %lx\n",
643             (void *)dhp, (void *)seg, (void *)addr, len));

645         /*
646          * Since the address space is "write" locked, we
647          * don't need the segment lock to protect "segdev" data.
648          */
649         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
649         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

651         if ((sz = sdp->softlockcnt) > 0) {
652                 /*
653                  * Fail the unmap if pages are SOFTLOCKed through this mapping.
654                  * softlockcnt is protected from change by the as write lock.
655                  */
656                 TRACE_1(TR_FAC_DEVMAP, TR_DEVMAP_UNMAP_CK1,
657                     "segdev_unmap:error softlockcnt = %ld", sz);
658                 DEBUGF(1, (CE_CONT, "segdev_unmap: softlockcnt %ld\n", sz));
659                 return (EAGAIN);
660         }

662         /*
663          * Check for bad sizes
664          */
665         if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
666             (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET))
667                 panic("segdev_unmap");

669         if (dhp != NULL) {
670                 devmap_handle_t *tdhp;
671                 /*
672                  * If large page size was used in hat_devload(),
673                  * the same page size must be used in hat_unload().
674                  */
675                 dhpp = tdhp = devmap_find_handle(dhp, addr);
676                 while (tdhp != NULL) {
677                         if (tdhp->dh_flags & DEVMAP_FLAG_LARGE) {
678                                 break;
679                         }
680                         tdhp = tdhp->dh_next;
681                 }
682                 if (tdhp != NULL) {     /* found a dhp using large pages */
683                         size_t slen = len;
684                         size_t mlen;
685                         size_t soff;

687                         soff = (ulong_t)(addr - dhpp->dh_uvaddr);
688                         while (slen != 0) {
689                                 mlen = MIN(slen, (dhpp->dh_len - soff));
690                                 hat_unload(seg->s_as->a_hat, dhpp->dh_uvaddr,
691                                     dhpp->dh_len, HAT_UNLOAD_UNMAP);
692                                 dhpp = dhpp->dh_next;
693                                 ASSERT(slen >= mlen);
694                                 slen -= mlen;
695                                 soff = 0;
696                         }
697                 } else
```

```
698                         hat_unload(seg->s_as->a_hat, addr, len,
699                             HAT_UNLOAD_UNMAP);
700         } else {
701                 /*
702                  * Unload any hardware translations in the range
703                  * to be taken out.
704                  */
705                 hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD_UNMAP);
706         }

708         /*
709          * get the user offset which will used in the driver callbacks
710          */
711         off = sdp->offset + (offset_t)(addr - seg->s_base);

713         /*
714          * Inform the vnode of the unmapping.
715          */
716         ASSERT(sdp->vp != NULL);
717         (void) VOP_DELMAP(VTOCVP(sdp->vp), off, seg->s_as, addr, len,
718             sdp->prot, sdp->maxprot, sdp->type, CRED(), NULL);

720         /*
721          * Check for entire segment
722          */
723         if (addr == seg->s_base && len == seg->s_size) {
724                 seg_free(seg);
725                 return (0);
726         }

728         opages = seg_pages(seg);
729         dpages = btop(len);
730         npages = opages - dpages;

732         /*
733          * Check for beginning of segment
734          */
735         if (addr == seg->s_base) {
736                 if (sdp->vpage != NULL) {
737                         register struct vpage *ovpage;

739                         ovpage = sdp->vpage;    /* keep pointer to vpage */

741                         nbytes = vpgtob(npages);
742                         sdp->vpage = kmem_alloc(nbytes, KM_SLEEP);
743                         bcopy(&ovpage[dpages], sdp->vpage, nbytes);

745                         /* free up old vpage */
746                         kmem_free(ovpage, vpgtob(opages));
747                 }

749                 /*
750                  * free devmap handles from the beginning of the mapping.
751                  */
752                 if (dhp != NULL)
753                         devmap_handle_unmap_head(dhp, len);

755                 sdp->offset += (offset_t)len;

757                 seg->s_base += len;
758                 seg->s_size -= len;

760                 return (0);
761         }

763         /*
```



```
764          * Check for end of segment
765          */
766         if (addr + len == seg->s_base + seg->s_size) {
767                 if (sdp->vpage != NULL) {
768                         register struct vpage *ovpage;

770                         ovpage = sdp->vpage;    /* keep pointer to vpage */

772                         nbytes = vpgtob(npages);
773                         sdp->vpage = kmem_alloc(nbytes, KM_SLEEP);
774                         bcopy(ovpage, sdp->vpage, nbytes);

776                         /* free up old vpage */
777                         kmem_free(ovpage, vpgtob(opages));
778                 }
779                 seg->s_size -= len;

781                 /*
782                  * free devmap handles from addr to the end of the mapping.
783                  */
784                 if (dhp != NULL)
785                         devmap_handle_unmap_tail(dhp, addr);

787                 return (0);
788         }

790         /*
791          * The section to go is in the middle of the segment,
792          * have to make it into two segments.  nseg is made for
793          * the high end while seg is cut down at the low end.
794          */
795         nbase = addr + len;                             /* new seg base */
796         nsize = (seg->s_base + seg->s_size) - nbase;    /* new seg size */
797         seg->s_size = addr - seg->s_base;               /* shrink old seg */
798         nseg = seg_alloc(seg->s_as, nbase, nsize);
799         if (nseg == NULL)
800                 panic("segdev_unmap seg_alloc");

802         TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_UNMAP_CK2,
803             "segdev_unmap: seg=%p nseg=%p", (void *)seg, (void *)nseg);
804         DEBUGF(3, (CE_CONT, "segdev_unmap: segdev_dup seg %p nseg %p\n",
805             (void *)seg, (void *)nseg));
806         nsdp = sdp_alloc();

808         nseg->s_ops = seg->s_ops;
809         nseg->s_data = (void *)nsdp;

811         VN_HOLD(sdp->vp);
812         nsdp->mapfunc = sdp->mapfunc;
813         nsdp->offset = sdp->offset + (offset_t)(nseg->s_base - seg->s_base);
814         nsdp->vp        = sdp->vp;
815         nsdp->pageprot = sdp->pageprot;
816         nsdp->prot      = sdp->prot;
817         nsdp->maxprot = sdp->maxprot;
818         nsdp->type = sdp->type;
819         nsdp->hat_attr = sdp->hat_attr;
820         nsdp->hat_flags = sdp->hat_flags;
821         nsdp->softlockcnt = 0;

823         /*
824          * Initialize per page data if the segment we are
825          * dup'ing has per page information.
826          */
827         if (sdp->vpage != NULL) {
828                 /* need to split vpage into two arrays */
829                 register size_t nnbytes;
```

```
830                 register size_t nnpages;
831                 register struct vpage *ovpage;

833                 ovpage = sdp->vpage;            /* keep pointer to vpage */

835                 npages = seg_pages(seg);        /* seg has shrunk */
836                 nbytes = vpgtob(npages);
837                 nnpages = seg_pages(nseg);
838                 nnbytes = vpgtob(nnpages);

840                 sdp->vpage = kmem_alloc(nbytes, KM_SLEEP);
841                 bcopy(ovpage, sdp->vpage, nbytes);

843                 nsdp->vpage = kmem_alloc(nnbytes, KM_SLEEP);
844                 bcopy(&ovpage[npages + dpages], nsdp->vpage, nnbytes);

846                 /* free up old vpage */
847                 kmem_free(ovpage, vpgtob(opages));
848         } else
849                 nsdp->vpage = NULL;

851         /*
852          * unmap dhps.
853          */
854         if (dhp == NULL) {
855                 nsdp->devmap_data = NULL;
856                 return (0);
857         }
858         while (dhp != NULL) {
859                 callbackops = &dhp->dh_callbackops;
860                 TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_UNMAP_CK3,
861                     "segdev_unmap: dhp=%p addr=%p", dhp, addr);
862                 DEBUGF(3, (CE_CONT, "unmap: dhp %p addr %p uvaddr %p len %lx\n",
863                     (void *)dhp, (void *)addr,
864                     (void *)dhp->dh_uvaddr, dhp->dh_len));

866                 if (addr == (dhp->dh_uvaddr + dhp->dh_len)) {
867                         dhpp = dhp->dh_next;
868                         dhp->dh_next = NULL;
869                         dhp = dhpp;
870                 } else if (addr > (dhp->dh_uvaddr + dhp->dh_len)) {
871                         dhp = dhp->dh_next;
872                 } else if (addr > dhp->dh_uvaddr &&
873                     (addr + len) < (dhp->dh_uvaddr + dhp->dh_len)) {
874                         /*
875                          * <addr, addr+len> is enclosed by dhp.
876                          * create a newdhp that begins at addr+len and
877                          * ends at dhp->dh_uvaddr+dhp->dh_len.
878                          */
879                         newdhp = kmem_alloc(sizeof (devmap_handle_t), KM_SLEEP);
880                         HOLD_DHP_LOCK(dhp);
881                         bcopy(dhp, newdhp, sizeof (devmap_handle_t));
882                         RELE_DHP_LOCK(dhp);
883                         newdhp->dh_seg = nseg;
884                         newdhp->dh_next = dhp->dh_next;
885                         if (dhp->dh_softlock != NULL)
886                                 newdhp->dh_softlock = devmap_softlock_init(
887                                     newdhp->dh_dev,
888                                     (ulong_t)callbackops->devmap_access);
889                         if (dhp->dh_ctx != NULL)
890                                 newdhp->dh_ctx = devmap_ctxinit(newdhp->dh_dev,
891                                     (ulong_t)callbackops->devmap_access);
892                         if (newdhp->dh_flags & DEVMAP_LOCK_INITED) {
893                                 mutex_init(&newdhp->dh_lock,
894                                     NULL, MUTEX_DEFAULT, NULL);
895                         }
```



```
896                         if (callbackops->devmap_unmap != NULL)
897                                 (*callbackops->devmap_unmap)(dhp, dhp->dh_pvtp,
898                                     off, len, dhp, &dhp->dh_pvtp,
899                                     newdhp, &newdhp->dh_pvtp);
900                         mlen = len + (addr - dhp->dh_uvaddr);
901                         devmap_handle_reduce_len(newdhp, mlen);
902                         nsdp->devmap_data = newdhp;
903                         /* XX Changing len should recalculate LARGE flag */
904                         dhp->dh_len = addr - dhp->dh_uvaddr;
905                         dhpp = dhp->dh_next;
906                         dhp->dh_next = NULL;
907                         dhp = dhpp;
908                 } else if ((addr > dhp->dh_uvaddr) &&
909                     ((addr + len) >= (dhp->dh_uvaddr + dhp->dh_len))) {
910                         mlen = dhp->dh_len + dhp->dh_uvaddr - addr;
911                         /*
912                          * <addr, addr+len> spans over dhps.
913                          */
914                         if (callbackops->devmap_unmap != NULL)
915                                 (*callbackops->devmap_unmap)(dhp, dhp->dh_pvtp,
916                                     off, mlen, (devmap_cookie_t *)dhp,
917                                     &dhp->dh_pvtp, NULL, NULL);
918                         /* XX Changing len should recalculate LARGE flag */
919                         dhp->dh_len = addr - dhp->dh_uvaddr;
920                         dhpp = dhp->dh_next;
921                         dhp->dh_next = NULL;
922                         dhp = dhpp;
923                         nsdp->devmap_data = dhp;
924                 } else if ((addr + len) >= (dhp->dh_uvaddr + dhp->dh_len)) {
925                         /*
926                          * dhp is enclosed by <addr, addr+len>.
927                          */
928                         dhp->dh_seg = nseg;
929                         nsdp->devmap_data = dhp;
930                         dhp = devmap_handle_unmap(dhp);
931                         nsdp->devmap_data = dhp; /* XX redundant? */
932                 } else if (((addr + len) > dhp->dh_uvaddr) &&
933                     ((addr + len) < (dhp->dh_uvaddr + dhp->dh_len))) {
934                         mlen = addr + len - dhp->dh_uvaddr;
935                         if (callbackops->devmap_unmap != NULL)
936                                 (*callbackops->devmap_unmap)(dhp, dhp->dh_pvtp,
937                                     dhp->dh_uoff, mlen, NULL,
938                                     NULL, dhp, &dhp->dh_pvtp);
939                         devmap_handle_reduce_len(dhp, mlen);
940                         nsdp->devmap_data = dhp;
941                         dhp->dh_seg = nseg;
942                         dhp = dhp->dh_next;
943                 } else {
944                         dhp->dh_seg = nseg;
945                         dhp = dhp->dh_next;
946                 }
947         }
948         return (0);
949 }
```

_____***unchanged_portion_omitted_***

```
1120 /*
1121  * Free a segment.
1122  */
1123 static void
1124 segdev_free(struct seg *seg)
1125 {
1126         register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
1127         devmap_handle_t *dhp = (devmap_handle_t *)sdp->devmap_data;

1129         TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_FREE,
```

```
1130             "segdev_free: dhp=%p seg=%p", (void *)dhp, (void *)seg);
1131         DEBUGF(3, (CE_CONT, "segdev_free: dhp %p seg %p\n",
1132             (void *)dhp, (void *)seg));

1134         /*
1135          * Since the address space is "write" locked, we
1136          * don't need the segment lock to protect "segdev" data.
1137          */
1138         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1138         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1140         while (dhp != NULL)
1141                 dhp = devmap_handle_unmap(dhp);

1143         VN_RELE(sdp->vp);
1144         if (sdp->vpage != NULL)
1145                 kmem_free(sdp->vpage, vpgtob(seg_pages(seg)));

1147         rw_destroy(&sdp->lock);
1148         kmem_free(sdp, sizeof (*sdp));
1149 }
______unchanged_portion_omitted_

1594 static faultcode_t
1595 segdev_fault(
1596         struct hat *hat,                /* the hat */
1597         struct seg *seg,                /* the seg_dev of interest */
1598         caddr_t addr,                   /* the address of the fault */
1599         size_t len,                     /* the length of the range */
1600         enum fault_type type,           /* type of fault */
1601         enum seg_rw rw)                 /* type of access at fault */
1602 {
1603         struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
1604         devmap_handle_t *dhp_head = (devmap_handle_t *)sdp->devmap_data;
1605         devmap_handle_t *dhp;
1606         struct devmap_softlock *slock = NULL;
1607         ulong_t slpage = 0;
1608         ulong_t off;
1609         caddr_t maddr = addr;
1610         int err;
1611         int err_is_faultcode = 0;

1613         TRACE_5(TR_FAC_DEVMAP, TR_DEVMAP_FAULT,
1614             "segdev_fault: dhp_head=%p seg=%p addr=%p len=%lx type=%x",
1615             (void *)dhp_head, (void *)seg, (void *)addr, len, type);
1616         DEBUGF(7, (CE_CONT, "segdev_fault: dhp_head %p seg %p "
1617             "addr %p len %lx type %x\n",
1618             (void *)dhp_head, (void *)seg, (void *)addr, len, type));

1620         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
1620         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1622         /* Handle non-devmap case */
1623         if (dhp_head == NULL)
1624                 return (segdev_faultpages(hat, seg, addr, len, type, rw, NULL));

1626         /* Find devmap handle */
1627         if ((dhp = devmap_find_handle(dhp_head, addr)) == NULL)
1628                 return (FC_NOMAP);

1630         /*
1631          * The seg_dev driver does not implement copy-on-write,
1632          * and always loads translations with maximal allowed permissions
1633          * but we got an fault trying to access the device.
1634          * Servicing the fault is not going to result in any better result
1635          * RFE: If we want devmap_access callbacks to be involved in F_PROT
```



```
1636          *      faults, then the code below is written for that
1637          *      Pending resolution of the following:
1638          *      - determine if the F_INVAL/F_SOFTLOCK syncing
1639          *      is needed for F_PROT also or not. The code below assumes it does
1640          *      - If driver sees F_PROT and calls devmap_load with same type,
1641          *      then segdev_faultpages will fail with FC_PROT anyway, need to
1642          *      change that so calls from devmap_load to segdev_faultpages for
1643          *      F_PROT type are retagged to F_INVAL.
1644          * RFE: Today we dont have drivers that use devmap and want to handle
1645          *      F_PROT calls. The code in segdev_fault* is written to allow
1646          *      this case but is not tested. A driver that needs this capability
1647          *      should be able to remove the short-circuit case; resolve the
1648          *      above issues and "should" work.
1649          */
1650         if (type == F_PROT) {
1651                 return (FC_PROT);
1652         }

1654         /*
1655          * Loop through dhp list calling devmap_access or segdev_faultpages for
1656          * each devmap handle.
1657          * drivers which implement devmap_access can interpose on faults and do
1658          * device-appropriate special actions before calling devmap_load.
1659          */

1661         /*
1662          * Unfortunately, this simple loop has turned out to expose a variety
1663          * of complex problems which results in the following convoluted code.
1664          *
1665          * First, a desire to handle a serialization of F_SOFTLOCK calls
1666          * to the driver within the framework.
1667          *      This results in a dh_softlock structure that is on a per device
1668          *      (or device instance) basis and serializes devmap_access calls.
1669          *      Ideally we would need to do this for underlying
1670          *      memory/device regions that are being faulted on
1671          *      but that is hard to identify and with REMAP, harder
1672          * Second, a desire to serialize F_INVAL(and F_PROT) calls w.r.t.
1673          *      to F_SOFTLOCK calls to the driver.
1674          * These serializations are to simplify the driver programmer model.
1675          * To support these two features, the code first goes through the
1676          *      devmap handles and counts the pages (slpage) that are covered
1677          *      by devmap_access callbacks.
1678          * This part ends with a devmap_softlock_enter call
1679          *      which allows only one F_SOFTLOCK active on a device instance,
1680          *      but multiple F_INVAL/F_PROTs can be active except when a
1681          *      F_SOFTLOCK is active
1682          *
1683          * Next, we dont short-circuit the fault code upfront to call
1684          *      segdev_softunlock for F_SOFTUNLOCK, because we must use
1685          *      the same length when we softlock and softunlock.
1686          *
1687          *      -Hat layers may not support softunlocking lengths less than the
1688          *      original length when there is large page support.
1689          *      -kpmem locking is dependent on keeping the lengths same.
1690          *      -if drivers handled F_SOFTLOCK, they probably also expect to
1691          *              see an F_SOFTUNLOCK of the same length
1692          *      Hence, if extending lengths during softlock,
1693          *      softunlock has to make the same adjustments and goes through
1694          *      the same loop calling segdev_faultpages/segdev_softunlock
1695          *      But some of the synchronization and error handling is different
1696          */

1698         if (type != F_SOFTUNLOCK) {
1699                 devmap_handle_t *dhpp = dhp;
1700                 size_t slen = len;
```

```
1702                /*
1703                 * Calculate count of pages that are :
1704                 * a) within the (potentially extended) fault region
1705                 * b) AND covered by devmap handle with devmap_access
1706                 */
1707                off = (ulong_t)(addr - dhpp->dh_uvaddr);
1708                while (slen != 0) {
1709                        size_t mlen;

1711                        /*
1712                         * Softlocking on a region that allows remap is
1713                         * unsupported due to unresolved locking issues
1714                         * XXX: unclear what these are?
1715                         *      One potential is that if there is a pending
1716                         *      softlock, then a remap should not be allowed
1717                         *      until the unlock is done. This is easily
1718                         *      fixed by returning error in devmap*remap on
1719                         *      checking the dh->dh_softlock->softlocked value
1720                         */
1721                        if ((type == F_SOFTLOCK) &&
1722                            (dhpp->dh_flags & DEVMAP_ALLOW_REMAP)) {
1723                                return (FC_NOSUPPORT);
1724                        }

1726                        mlen = MIN(slen, (dhpp->dh_len - off));
1727                        if (dhpp->dh_callbackops.devmap_access) {
1728                                size_t llen;
1729                                caddr_t laddr;
1730                                /*
1731                                 * use extended length for large page mappings
1732                                 */
1733                                HOLD_DHP_LOCK(dhpp);
1734                                if ((sdp->pageprot == 0) &&
1735                                    (dhpp->dh_flags & DEVMAP_FLAG_LARGE)) {
1736                                        devmap_get_large_pgsize(dhpp,
1737                                            mlen, maddr, &llen, &laddr);
1738                                } else {
1739                                        llen = mlen;
1740                                }
1741                                RELE_DHP_LOCK(dhpp);

1743                                slpage += btopr(llen);
1744                                slock = dhpp->dh_softlock;
1745                        }
1746                        maddr += mlen;
1747                        ASSERT(slen >= mlen);
1748                        slen -= mlen;
1749                        dhpp = dhpp->dh_next;
1750                        off = 0;
1751                }
1752                /*
1753                 * synchonize with other faulting threads and wait till safe
1754                 * devmap_softlock_enter might return due to signal in cv_wait
1755                 *
1756                 * devmap_softlock_enter has to be called outside of while loop
1757                 * to prevent a deadlock if len spans over multiple dhps.
1758                 * dh_softlock is based on device instance and if multiple dhps
1759                 * use the same device instance, the second dhp's LOCK call
1760                 * will hang waiting on the first to complete.
1761                 * devmap_setup verifies that slocks in a dhp_chain are same.
1762                 * RFE: this deadlock only hold true for F_SOFTLOCK. For
1763                 *      F_INVAL/F_PROT, since we now allow multiple in parallel,
1764                 *      we could have done the softlock_enter inside the loop
1765                 *      and supported multi-dhp mappings with dissimilar devices
1766                 */
1767                if (err = devmap_softlock_enter(slock, slpage, type))
```



```
1768                        return (FC_MAKE_ERR(err));
1769        }

1771        /* reset 'maddr' to the start addr of the range of fault. */
1772        maddr = addr;

1774        /* calculate the offset corresponds to 'addr' in the first dhp. */
1775        off = (ulong_t)(addr - dhp->dh_uvaddr);

1777        /*
1778         * The fault length may span over multiple dhps.
1779         * Loop until the total length is satisfied.
1780         */
1781        while (len != 0) {
1782                size_t llen;
1783                size_t mlen;
1784                caddr_t laddr;

1786                /*
1787                 * mlen is the smaller of 'len' and the length
1788                 * from addr to the end of mapping defined by dhp.
1789                 */
1790                mlen = MIN(len, (dhp->dh_len - off));

1792                HOLD_DHP_LOCK(dhp);
1793                /*
1794                 * Pass the extended length and address to devmap_access
1795                 * if large pagesize is used for loading address translations.
1796                 */
1797                if ((sdp->pageprot == 0) &&
1798                    (dhp->dh_flags & DEVMAP_FLAG_LARGE)) {
1799                        devmap_get_large_pgsize(dhp, mlen, maddr,
1800                            &llen, &laddr);
1801                        ASSERT(maddr == addr || laddr == maddr);
1802                } else {
1803                        llen = mlen;
1804                        laddr = maddr;
1805                }

1807                if (dhp->dh_callbackops.devmap_access != NULL) {
1808                        offset_t aoff;

1810                        aoff = sdp->offset + (offset_t)(laddr - seg->s_base);

1812                        /*
1813                         * call driver's devmap_access entry point which will
1814                         * call devmap_load/contextmgmt to load the translations
1815                         *
1816                         * We drop the dhp_lock before calling access so
1817                         * drivers can call devmap_*_remap within access
1818                         */
1819                        RELE_DHP_LOCK(dhp);

1821                        err = (*dhp->dh_callbackops.devmap_access)(
1822                            dhp, (void *)dhp->dh_pvtp, aoff, llen, type, rw);
1823                } else {
1824                        /*
1825                         * If no devmap_access entry point, then load mappings
1826                         * hold dhp_lock across faultpages if REMAP
1827                         */
1828                        err = segdev_faultpages(hat, seg, laddr, llen,
1829                            type, rw, dhp);
1830                        err_is_faultcode = 1;
1831                        RELE_DHP_LOCK(dhp);
1832                }
```

```
1834                 if (err) {
1835                         if ((type == F_SOFTLOCK) && (maddr > addr)) {
1836                                 /*
1837                                  * If not first dhp, use
1838                                  * segdev_fault(F_SOFTUNLOCK) for prior dhps
1839                                  * While this is recursion, it is incorrect to
1840                                  * call just segdev_softunlock
1841                                  * if we are using either large pages
1842                                  * or devmap_access. It will be more right
1843                                  * to go through the same loop as above
1844                                  * rather than call segdev_softunlock directly
1845                                  * It will use the right lenghths as well as
1846                                  * call into the driver devmap_access routines.
1847                                  */
1848                                 size_t done = (size_t)(maddr - addr);
1849                                 (void) segdev_fault(hat, seg, addr, done,
1850                                     F_SOFTUNLOCK, S_OTHER);
1851                                 /*
1852                                  * reduce slpage by number of pages
1853                                  * released by segdev_softunlock
1854                                  */
1855                                 ASSERT(slpage >= btopr(done));
1856                                 devmap_softlock_exit(slock,
1857                                     slpage - btopr(done), type);
1858                         } else {
1859                                 devmap_softlock_exit(slock, slpage, type);
1860                         }


1863                         /*
1864                          * Segdev_faultpages() already returns a faultcode,
1865                          * hence, result from segdev_faultpages() should be
1866                          * returned directly.
1867                          */
1868                         if (err_is_faultcode)
1869                                 return (err);
1870                         return (FC_MAKE_ERR(err));
1871                 }

1873                 maddr += mlen;
1874                 ASSERT(len >= mlen);
1875                 len -= mlen;
1876                 dhp = dhp->dh_next;
1877                 off = 0;

1879                 ASSERT(!dhp || len == 0 || maddr == dhp->dh_uvaddr);
1880         }
1881         /*
1882          * release the softlock count at end of fault
1883          * For F_SOFTLOCk this is done in the later F_SOFTUNLOCK
1884          */
1885         if ((type == F_INVAL) || (type == F_PROT))
1886                 devmap_softlock_exit(slock, slpage, type);
1887         return (0);
1888 }
```

*_____unchanged_portion_omitted_*

```
2050 /*
2051  * Asynchronous page fault.  We simply do nothing since this
2052  * entry point is not supposed to load up the translation.
2053  */
2054 /*ARGSUSED*/
2055 static faultcode_t
2056 segdev_faulta(struct seg *seg, caddr_t addr)
2057 {
2058         TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_FAULTA,
```



```
2059             "segdev_faulta: seg=%p addr=%p", (void *)seg, (void *)addr);
2060         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2060         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2062         return (0);
2063 }

2065 static int
2066 segdev_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
2067 {
2068         register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
2069         register devmap_handle_t *dhp;
2070         register struct vpage *vp, *evp;
2071         devmap_handle_t *dhp_head = (devmap_handle_t *)sdp->devmap_data;
2072         ulong_t off;
2073         size_t mlen, sz;

2075         TRACE_4(TR_FAC_DEVMAP, TR_DEVMAP_SETPROT,
2076             "segdev_setprot:start seg=%p addr=%p len=%lx prot=%x",
2077             (void *)seg, (void *)addr, len, prot);
2078         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2078         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2080         if ((sz = sdp->softlockcnt) > 0 && dhp_head != NULL) {
2081                 /*
2082                  * Fail the setprot if pages are SOFTLOCKed through this
2083                  * mapping.
2084                  * Softlockcnt is protected from change by the as read lock.
2085                  */
2086                 TRACE_1(TR_FAC_DEVMAP, TR_DEVMAP_SETPROT_CK1,
2087                     "segdev_setprot:error softlockcnt=%lx", sz);
2088                 DEBUGF(1, (CE_CONT, "segdev_setprot: softlockcnt %ld\n", sz));
2089                 return (EAGAIN);
2090         }

2092         if (dhp_head != NULL) {
2093                 if ((dhp = devmap_find_handle(dhp_head, addr)) == NULL)
2094                         return (EINVAL);

2096                 /*
2097                  * check if violate maxprot.
2098                  */
2099                 off = (ulong_t)(addr - dhp->dh_uvaddr);
2100                 mlen  = len;
2101                 while (dhp) {
2102                         if ((dhp->dh_maxprot & prot) != prot)
2103                                 return (EACCES);        /* violated maxprot */

2105                         if (mlen > (dhp->dh_len - off)) {
2106                                 mlen -= dhp->dh_len - off;
2107                                 dhp = dhp->dh_next;
2108                                 off = 0;
2109                         } else
2110                                 break;
2111                 }
2112         } else {
2113                 if ((sdp->maxprot & prot) != prot)
2114                         return (EACCES);
2115         }

2117         rw_enter(&sdp->lock, RW_WRITER);
2118         if (addr == seg->s_base && len == seg->s_size && sdp->pageprot == 0) {
2119                 if (sdp->prot == prot) {
2120                         rw_exit(&sdp->lock);
2121                         return (0);                     /* all done */
2122                 }
```

```
2123                 sdp->prot = (uchar_t)prot;
2124         } else {
2125                 sdp->pageprot = 1;
2126                 if (sdp->vpage == NULL) {
2127                         /*
2128                          * First time through setting per page permissions,
2129                          * initialize all the vpage structures to prot
2130                          */
2131                         sdp->vpage = kmem_zalloc(vpgtob(seg_pages(seg)),
2132                             KM_SLEEP);
2133                         evp = &sdp->vpage[seg_pages(seg)];
2134                         for (vp = sdp->vpage; vp < evp; vp++)
2135                                 VPP_SETPROT(vp, sdp->prot);
2136                 }
2137                 /*
2138                  * Now go change the needed vpages protections.
2139                  */
2140                 evp = &sdp->vpage[seg_page(seg, addr + len)];
2141                 for (vp = &sdp->vpage[seg_page(seg, addr)]; vp < evp; vp++)
2142                         VPP_SETPROT(vp, prot);
2143         }
2144         rw_exit(&sdp->lock);

2146         if (dhp_head != NULL) {
2147                 devmap_handle_t *tdhp;
2148                 /*
2149                  * If large page size was used in hat_devload(),
2150                  * the same page size must be used in hat_unload().
2151                  */
2152                 dhp = tdhp = devmap_find_handle(dhp_head, addr);
2153                 while (tdhp != NULL) {
2154                         if (tdhp->dh_flags & DEVMAP_FLAG_LARGE) {
2155                                 break;
2156                         }
2157                         tdhp = tdhp->dh_next;
2158                 }
2159                 if (tdhp) {
2160                         size_t slen = len;
2161                         size_t mlen;
2162                         size_t soff;

2164                         soff = (ulong_t)(addr - dhp->dh_uvaddr);
2165                         while (slen != 0) {
2166                                 mlen = MIN(slen, (dhp->dh_len - soff));
2167                                 hat_unload(seg->s_as->a_hat, dhp->dh_uvaddr,
2168                                     dhp->dh_len, HAT_UNLOAD);
2169                                 dhp = dhp->dh_next;
2170                                 ASSERT(slen >= mlen);
2171                                 slen -= mlen;
2172                                 soff = 0;
2173                         }
2174                         return (0);
2175                 }
2176         }

2178         if ((prot & ~PROT_USER) == PROT_NONE) {
2179                 hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
2180         } else {
2181                 /*
2182                  * RFE: the segment should keep track of all attributes
2183                  * allowing us to remove the deprecated hat_chgprot
2184                  * and use hat_chgattr.
2185                  */
2186                 hat_chgprot(seg->s_as->a_hat, addr, len, prot);
2187         }
```



```
2189         return (0);
2190 }

2192 static int
2193 segdev_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
2194 {
2195         struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
2196         struct vpage *vp, *evp;

2198         TRACE_4(TR_FAC_DEVMAP, TR_DEVMAP_CHECKPROT,
2199             "segdev_checkprot:start seg=%p addr=%p len=%lx prot=%x",
2200             (void *)seg, (void *)addr, len, prot);
2201         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2201         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2203         /*
2204          * If segment protection can be used, simply check against them
2205          */
2206         rw_enter(&sdp->lock, RW_READER);
2207         if (sdp->pageprot == 0) {
2208                 register int err;

2210                 err = ((sdp->prot & prot) != prot) ? EACCES : 0;
2211                 rw_exit(&sdp->lock);
2212                 return (err);
2213         }

2215         /*
2216          * Have to check down to the vpage level
2217          */
2218         evp = &sdp->vpage[seg_page(seg, addr + len)];
2219         for (vp = &sdp->vpage[seg_page(seg, addr)]; vp < evp; vp++) {
2220                 if ((VPP_PROT(vp) & prot) != prot) {
2221                         rw_exit(&sdp->lock);
2222                         return (EACCES);
2223                 }
2224         }
2225         rw_exit(&sdp->lock);
2226         return (0);
2227 }

2229 static int
2230 segdev_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
2231 {
2232         struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
2233         size_t pgno;

2235         TRACE_4(TR_FAC_DEVMAP, TR_DEVMAP_GETPROT,
2236             "segdev_getprot:start seg=%p addr=%p len=%lx protv=%p",
2237             (void *)seg, (void *)addr, len, (void *)protv);
2238         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2238         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2240         pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;
2241         if (pgno != 0) {
2242                 rw_enter(&sdp->lock, RW_READER);
2243                 if (sdp->pageprot == 0) {
2244                         do {
2245                                 protv[--pgno] = sdp->prot;
2246                         } while (pgno != 0);
2247                 } else {
2248                         size_t pgoff = seg_page(seg, addr);

2250                         do {
2251                                 pgno--;
2252                                 protv[pgno] =
```

```
2253                                    VPP_PROT(&sdp->vpage[pgno + pgoff]);
2254                            } while (pgno != 0);
2255                    }
2256                    rw_exit(&sdp->lock);
2257            }
2258            return (0);
2259    }

2261    static u_offset_t
2262    segdev_getoffset(register struct seg *seg, caddr_t addr)
2263    {
2264            register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;

2266            TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_GETOFFSET,
2267                "segdev_getoffset:start seg=%p addr=%p", (void *)seg, (void *)addr);

2269            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2269            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2271            return ((u_offset_t)sdp->offset + (addr - seg->s_base));
2272    }

2274    /*ARGSUSED*/
2275    static int
2276    segdev_gettype(register struct seg *seg, caddr_t addr)
2277    {
2278            register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;

2280            TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_GETTYPE,
2281                "segdev_gettype:start seg=%p addr=%p", (void *)seg, (void *)addr);

2283            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2283            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2285            return (sdp->type);
2286    }


2289    /*ARGSUSED*/
2290    static int
2291    segdev_getvp(register struct seg *seg, caddr_t addr, struct vnode **vpp)
2292    {
2293            register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;

2295            TRACE_2(TR_FAC_DEVMAP, TR_DEVMAP_GETVP,
2296                "segdev_getvp:start seg=%p addr=%p", (void *)seg, (void *)addr);

2298            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2298            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2300            /*
2301             * Note that this vp is the common_vp of the device, where the
2302             * pages are hung ..
2303             */
2304            *vpp = VTOCVP(sdp->vp);

2306            return (0);
2307    }
```

_____***unchanged_portion_omitted_***

```
2318    /*
2319     * segdev pages are not in the cache, and thus can't really be controlled.
2320     * Hence, syncs are simply always successful.
2321     */
2322    /*ARGSUSED*/
2323    static int
```



```
2324    segdev_sync(struct seg *seg, caddr_t addr, size_t len, int attr, uint_t flags)
2325    {
2326            TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_SYNC, "segdev_sync:start");

2328            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2328            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2330            return (0);
2331    }

2333    /*
2334     * segdev pages are always "in core".
2335     */
2336    /*ARGSUSED*/
2337    static size_t
2338    segdev_incore(struct seg *seg, caddr_t addr, size_t len, char *vec)
2339    {
2340            size_t v = 0;

2342            TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_INCORE, "segdev_incore:start");

2344            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2344            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2346            for (len = (len + PAGEOFFSET) & PAGEMASK; len; len -= PAGESIZE,
2347                v += PAGESIZE)
2348                    *vec++ = 1;
2349            return (v);
2350    }

2352    /*
2353     * segdev pages are not in the cache, and thus can't really be controlled.
2354     * Hence, locks are simply always successful.
2355     */
2356    /*ARGSUSED*/
2357    static int
2358    segdev_lockop(struct seg *seg, caddr_t addr,
2359        size_t len, int attr, int op, ulong_t *lockmap, size_t pos)
2360    {
2361            TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_LOCKOP, "segdev_lockop:start");

2363            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2363            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2365            return (0);
2366    }

2368    /*
2369     * segdev pages are not in the cache, and thus can't really be controlled.
2370     * Hence, advise is simply always successful.
2371     */
2372    /*ARGSUSED*/
2373    static int
2374    segdev_advise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
2375    {
2376            TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_ADVISE, "segdev_advise:start");

2378            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2378            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2380            return (0);
2381    }
```

_____***unchanged_portion_omitted_***

```
3054    /*
3055     * devmap_load:
```

```
3056  *                     Marks a segdev segment or pages if offset->offset+len
3057  *                     is not the entire segment as nointercept and faults in
3058  *                     the pages in the range offset -> offset+len.
3059  */
3060 int
3061 devmap_load(devmap_cookie_t dhc, offset_t offset, size_t len, uint_t type,
3062     uint_t rw)
3063 {
3064         devmap_handle_t *dhp = (devmap_handle_t *)dhc;
3065         struct as *asp = dhp->dh_seg->s_as;
3066         caddr_t addr;
3067         ulong_t size;
3068         ssize_t soff;   /* offset from the beginning of the segment */
3069         int rc;

3071         TRACE_3(TR_FAC_DEVMAP, TR_DEVMAP_LOAD,
3072             "devmap_load:start dhp=%p offset=%llx len=%lx",
3073             (void *)dhp, offset, len);

3075         DEBUGF(7, (CE_CONT, "devmap_load: dhp %p offset %llx len %lx\n",
3076             (void *)dhp, offset, len));

3078         /*
3079          *      Hat layer only supports devload to process' context for which
3080          *      the as lock is held. Verify here and return error if drivers
3081          *      inadvertently call devmap_load on a wrong devmap handle.
3082          */
3083         if ((asp != &kas) && !AS_LOCK_HELD(asp))
3083         if ((asp != &kas) && !AS_LOCK_HELD(asp, &asp->a_lock))
3084                 return (FC_MAKE_ERR(EINVAL));

3086         soff = (ssize_t)(offset - dhp->dh_uoff);
3087         soff = round_down_p2(soff, PAGESIZE);
3088         if (soff < 0 || soff >= dhp->dh_len)
3089                 return (FC_MAKE_ERR(EINVAL));

3091         /*
3092          * Address and size must be page aligned.  Len is set to the
3093          * number of bytes in the number of pages that are required to
3094          * support len.  Offset is set to the byte offset of the first byte
3095          * of the page that contains offset.
3096          */
3097         len = round_up_p2(len, PAGESIZE);

3099         /*
3100          * If len == 0, then calculate the size by getting
3101          * the number of bytes from offset to the end of the segment.
3102          */
3103         if (len == 0)
3104                 size = dhp->dh_len - soff;
3105         else {
3106                 size = len;
3107                 if ((soff + size) > dhp->dh_len)
3108                         return (FC_MAKE_ERR(EINVAL));
3109         }

3111         /*
3112          * The address is offset bytes from the base address of
3113          * the segment.
3114          */
3115         addr = (caddr_t)(soff + dhp->dh_uvaddr);

3117         HOLD_DHP_LOCK(dhp);
3118         rc = segdev_faultpages(asp->a_hat,
3119             dhp->dh_seg, addr, size, type, rw, dhp);
3120         RELE_DHP_LOCK(dhp);
```



```
3121         return (rc);
3122 }
_____unchanged_portion_omitted_
```

****
9899 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/vm/seg_kpm.c
patch as-lock-macro-simplification
****
_____***unchanged_portion_omitted_***

```
 204 /*
 205  * This routine is called via a machine specific fault handling
 206  * routine.
 207  */
 208 /* ARGSUSED */
 209 faultcode_t
 210 segkpm_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
 211         enum fault_type type, enum seg_rw rw)
 212 {
 213         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 213         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

 215         switch (type) {
 216         case F_INVAL:
 217                 return (hat_kpm_fault(hat, addr));
 218         case F_SOFTLOCK:
 219         case F_SOFTUNLOCK:
 220                 return (0);
 221         default:
 222                 return (FC_NOSUPPORT);
 223         }
 224         /*NOTREACHED*/
 225 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   58066 Wed Nov 25 13:59:39 2015
new/usr/src/uts/common/vm/seg_map.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 850 static int
 851 segmap_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
 852 {
 853         struct segmap_data *smd = (struct segmap_data *)seg->s_data;
 854         size_t pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;

 856         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 856         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

 858         if (pgno != 0) {
 859                 do {
 860                         protv[--pgno] = smd->smd_prot;
 861                 } while (pgno != 0);
 862         }
 863         return (0);
 864 }
______unchanged_portion_omitted_
```

```
**********************************************************
   83298 Wed Nov 25 13:59:40 2015
new/usr/src/uts/common/vm/seg_spt.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 234 /*
 235  * called from seg_free().
 236  * free (i.e., unlock, unmap, return to free list)
 237  *  all the pages in the given seg.
 238  */
 239 void
 240 segspt_free(struct seg  *seg)
 241 {
 242         struct spt_data *sptd = (struct spt_data *)seg->s_data;

 244         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
 244         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 246         if (sptd != NULL) {
 247                 if (sptd->spt_realsize)
 248                         segspt_free_pages(seg, seg->s_base, sptd->spt_realsize);

 250         if (sptd->spt_ppa_lckcnt)
 251                 kmem_free(sptd->spt_ppa_lckcnt,
 252                     sizeof (*sptd->spt_ppa_lckcnt)
 253                     * btopr(sptd->spt_amp->size));
 254                 kmem_free(sptd->spt_vp, sizeof (*sptd->spt_vp));
 255                 cv_destroy(&sptd->spt_cv);
 256                 mutex_destroy(&sptd->spt_lock);
 257                 kmem_free(sptd, sizeof (*sptd));
 258         }
 259 }

 261 /*ARGSUSED*/
 262 static int
 263 segspt_shmsync(struct seg *seg, caddr_t addr, size_t len, int attr,
 264         uint_t flags)
 265 {
 266         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 266         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

 268         return (0);
 269 }

 271 /*ARGSUSED*/
 272 static size_t
 273 segspt_shmincore(struct seg *seg, caddr_t addr, size_t len, char *vec)
 274 {
 275         caddr_t eo_seg;
 276         pgcnt_t npages;
 277         struct shm_data *shmd = (struct shm_data *)seg->s_data;
 278         struct seg      *sptseg;
 279         struct spt_data *sptd;

 281         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 281         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
 282 #ifdef lint
 283         seg = seg;
 284 #endif
 285         sptseg = shmd->shm_sptseg;
 286         sptd = sptseg->s_data;

 288         if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
 289                 eo_seg = addr + len;
```



```
 290                 while (addr < eo_seg) {
 291                         /* page exists, and it's locked. */
 292                         *vec++ = SEG_PAGE_INCORE | SEG_PAGE_LOCKED |
 293                             SEG_PAGE_ANON;
 294                         addr += PAGESIZE;
 295                 }
 296                 return (len);
 297         } else {
 298                 struct  anon_map *amp = shmd->shm_amp;
 299                 struct  anon    *ap;
 300                 page_t          *pp;
 301                 pgcnt_t         anon_index;
 302                 struct vnode    *vp;
 303                 u_offset_t      off;
 304                 ulong_t         i;
 305                 int             ret;
 306                 anon_sync_obj_t cookie;

 308                 addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
 309                 anon_index = seg_page(seg, addr);
 310                 npages = btopr(len);
 311                 if (anon_index + npages > btopr(shmd->shm_amp->size)) {
 312                         return (EINVAL);
 313                 }
 314                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
 315                 for (i = 0; i < npages; i++, anon_index++) {
 316                         ret = 0;
 317                         anon_array_enter(amp, anon_index, &cookie);
 318                         ap = anon_get_ptr(amp->ahp, anon_index);
 319                         if (ap != NULL) {
 320                                 swap_xlate(ap, &vp, &off);
 321                                 anon_array_exit(&cookie);
 322                                 pp = page_lookup_nowait(vp, off, SE_SHARED);
 323                                 if (pp != NULL) {
 324                                         ret |= SEG_PAGE_INCORE | SEG_PAGE_ANON;
 325                                         page_unlock(pp);
 326                                 }
 327                         } else {
 328                                 anon_array_exit(&cookie);
 329                         }
 330                         if (shmd->shm_vpage[anon_index] & DISM_PG_LOCKED) {
 331                                 ret |= SEG_PAGE_LOCKED;
 332                         }
 333                         *vec++ = (char)ret;
 334                 }
 335                 ANON_LOCK_EXIT(&amp->a_rwlock);
 336                 return (len);
 337         }
 338 }

 340 static int
 341 segspt_unmap(struct seg *seg, caddr_t raddr, size_t ssize)
 342 {
 343         size_t share_size;

 345         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
 345         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 347         /*
 348          * seg.s_size may have been rounded up to the largest page size
 349          * in shmat().
 350          * XXX This should be cleanedup. sptdestroy should take a length
 351          * argument which should be the same as sptcreate. Then
 352          * this rounding would not be needed (or is done in shm.c)
 353          * Only the check for full segment will be needed.
 354          *
```

```
355          * XXX -- shouldn't raddr == 0 always? These tests don't seem
356          * to be useful at all.
357          */
358         share_size = page_get_pagesize(seg->s_szc);
359         ssize = P2ROUNDUP(ssize, share_size);
360
361         if (raddr == seg->s_base && ssize == seg->s_size) {
362                 seg_free(seg);
363                 return (0);
364         } else
365                 return (EINVAL);
366 }
367
368 int
369 segspt_create(struct seg *seg, caddr_t argsp)
370 {
371         int             err;
372         caddr_t         addr = seg->s_base;
373         struct spt_data *sptd;
374         struct  segspt_crargs *sptcargs = (struct segspt_crargs *)argsp;
375         struct anon_map *amp = sptcargs->amp;
376         struct kshmid   *sp = amp->a_sp;
377         struct  cred    *cred = CRED();
378         ulong_t         i, j, anon_index = 0;
379         pgcnt_t         npages = btopr(amp->size);
380         struct vnode    *vp;
381         page_t          **ppa;
382         uint_t          hat_flags;
383         size_t          pgsz;
384         pgcnt_t         pgcnt;
385         caddr_t         a;
386         pgcnt_t         pidx;
387         size_t          sz;
388         proc_t          *procp = curproc;
389         rctl_qty_t      lockedbytes = 0;
390         kproject_t      *proj;
391
392         /*
393          * We are holding the a_lock on the underlying dummy as,
394          * so we can make calls to the HAT layer.
395          */
396         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
396         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
397         ASSERT(sp != NULL);
398
399 #ifdef DEBUG
400         TNF_PROBE_2(segspt_create, "spt", /* CSTYLED */,
401             tnf_opaque, addr, addr, tnf_ulong, len, seg->s_size);
402 #endif
403         if ((sptcargs->flags & SHM_PAGEABLE) == 0) {
404                 if (err = anon_swap_adjust(npages))
405                         return (err);
406         }
407         err = ENOMEM;
408
409         if ((sptd = kmem_zalloc(sizeof (*sptd), KM_NOSLEEP)) == NULL)
410                 goto out1;
411
412         if ((sptcargs->flags & SHM_PAGEABLE) == 0) {
413                 if ((ppa = kmem_zalloc(((sizeof (page_t *)) * npages),
414                     KM_NOSLEEP)) == NULL)
415                         goto out2;
416         }
417
418         mutex_init(&sptd->spt_lock, NULL, MUTEX_DEFAULT, NULL);
```



```
420         if ((vp = kmem_zalloc(sizeof (*vp), KM_NOSLEEP)) == NULL)
421                 goto out3;
422
423         seg->s_ops = &segspt_ops;
424         sptd->spt_vp = vp;
425         sptd->spt_amp = amp;
426         sptd->spt_prot = sptcargs->prot;
427         sptd->spt_flags = sptcargs->flags;
428         seg->s_data = (caddr_t)sptd;
429         sptd->spt_ppa = NULL;
430         sptd->spt_ppa_lckcnt = NULL;
431         seg->s_szc = sptcargs->szc;
432         cv_init(&sptd->spt_cv, NULL, CV_DEFAULT, NULL);
433         sptd->spt_gen = 0;
434
435         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
436         if (seg->s_szc > amp->a_szc) {
437                 amp->a_szc = seg->s_szc;
438         }
439         ANON_LOCK_EXIT(&amp->a_rwlock);
440
441         /*
442          * Set policy to affect initial allocation of pages in
443          * anon_map_createpages()
444          */
445         (void) lgrp_shm_policy_set(LGRP_MEM_POLICY_DEFAULT, amp, anon_index,
446             NULL, 0, ptob(npages));
447
448         if (sptcargs->flags & SHM_PAGEABLE) {
449                 size_t  share_sz;
450                 pgcnt_t new_npgs, more_pgs;
451                 struct anon_hdr *nahp;
452                 zone_t *zone;
453
454                 share_sz = page_get_pagesize(seg->s_szc);
455                 if (!IS_P2ALIGNED(amp->size, share_sz)) {
456                         /*
457                          * We are rounding up the size of the anon array
458                          * on 4 M boundary because we always create 4 M
459                          * of page(s) when locking, faulting pages and we
460                          * don't have to check for all corner cases e.g.
461                          * if there is enough space to allocate 4 M
462                          * page.
463                          */
464                         new_npgs = btop(P2ROUNDUP(amp->size, share_sz));
465                         more_pgs = new_npgs - npages;
466
467                         /*
468                          * The zone will never be NULL, as a fully created
469                          * shm always has an owning zone.
470                          */
471                         zone = sp->shm_perm.ipc_zone_ref.zref_zone;
472                         ASSERT(zone != NULL);
473                         if (anon_resv_zone(ptob(more_pgs), zone) == 0) {
474                                 err = ENOMEM;
475                                 goto out4;
476                         }
477
478                         nahp = anon_create(new_npgs, ANON_SLEEP);
479                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
480                         (void) anon_copy_ptr(amp->ahp, 0, nahp, 0, npages,
481                             ANON_SLEEP);
482                         anon_release(amp->ahp, npages);
483                         amp->ahp = nahp;
484                         ASSERT(amp->swresv == ptob(npages));
485                         amp->swresv = amp->size = ptob(new_npgs);
```

```
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        npages = new_npgs;
                }

                sptd->spt_ppa_lckcnt = kmem_zalloc(npages *
                    sizeof (*sptd->spt_ppa_lckcnt), KM_SLEEP);
                sptd->spt_pcachecnt = 0;
                sptd->spt_realsize = ptob(npages);
                sptcargs->seg_spt = seg;
                return (0);
        }

        /*
         * get array of pages for each anon slot in amp
         */
        if ((err = anon_map_createpages(amp, anon_index, ptob(npages), ppa,
            seg, addr, S_CREATE, cred)) != 0)
                goto out4;

        mutex_enter(&sp->shm_mlock);

        /* May be partially locked, so, count bytes to charge for locking */
        for (i = 0; i < npages; i++)
                if (ppa[i]->p_lckcnt == 0)
                        lockedbytes += PAGESIZE;

        proj = sp->shm_perm.ipc_proj;

        if (lockedbytes > 0) {
                mutex_enter(&procp->p_lock);
                if (rctl_incr_locked_mem(procp, proj, lockedbytes, 0)) {
                        mutex_exit(&procp->p_lock);
                        mutex_exit(&sp->shm_mlock);
                        for (i = 0; i < npages; i++)
                                page_unlock(ppa[i]);
                        err = ENOMEM;
                        goto out4;
                }
                mutex_exit(&procp->p_lock);
        }

        /*
         * addr is initial address corresponding to the first page on ppa list
         */
        for (i = 0; i < npages; i++) {
                /* attempt to lock all pages */
                if (page_pp_lock(ppa[i], 0, 1) == 0) {
                        /*
                         * if unable to lock any page, unlock all
                         * of them and return error
                         */
                        for (j = 0; j < i; j++)
                                page_pp_unlock(ppa[j], 0, 1);
                        for (i = 0; i < npages; i++)
                                page_unlock(ppa[i]);
                        rctl_decr_locked_mem(NULL, proj, lockedbytes, 0);
                        mutex_exit(&sp->shm_mlock);
                        err = ENOMEM;
                        goto out4;
                }
        }
        mutex_exit(&sp->shm_mlock);

        /*
         * Some platforms assume that ISM mappings are HAT_LOAD_LOCK
         * for the entire life of the segment. For example platforms
```



```
         * that do not support Dynamic Reconfiguration.
         */
        hat_flags = HAT_LOAD_SHARE;
        if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, NULL))
                hat_flags |= HAT_LOAD_LOCK;

        /*
         * Load translations one lare page at a time
         * to make sure we don't create mappings bigger than
         * segment's size code in case underlying pages
         * are shared with segvn's segment that uses bigger
         * size code than we do.
         */
        pgsz = page_get_pagesize(seg->s_szc);
        pgcnt = page_get_pagecnt(seg->s_szc);
        for (a = addr, pidx = 0; pidx < npages; a += pgsz, pidx += pgcnt) {
                sz = MIN(pgsz, ptob(npages - pidx));
                hat_memload_array(seg->s_as->a_hat, a, sz,
                    &ppa[pidx], sptd->spt_prot, hat_flags);
        }

        /*
         * On platforms that do not support HAT_DYNAMIC_ISM_UNMAP,
         * we will leave the pages locked SE_SHARED for the life
         * of the ISM segment. This will prevent any calls to
         * hat_pageunload() on this ISM segment for those platforms.
         */
        if (!(hat_flags & HAT_LOAD_LOCK)) {
                /*
                 * On platforms that support HAT_DYNAMIC_ISM_UNMAP,
                 * we no longer need to hold the SE_SHARED lock on the pages,
                 * since L_PAGELOCK and F_SOFTLOCK calls will grab the
                 * SE_SHARED lock on the pages as necessary.
                 */
                for (i = 0; i < npages; i++)
                        page_unlock(ppa[i]);
        }
        sptd->spt_pcachecnt = 0;
        kmem_free(ppa, ((sizeof (page_t *)) * npages));
        sptd->spt_realsize = ptob(npages);
        atomic_add_long(&spt_used, npages);
        sptcargs->seg_spt = seg;
        return (0);

out4:
        seg->s_data = NULL;
        kmem_free(vp, sizeof (*vp));
        cv_destroy(&sptd->spt_cv);
out3:
        mutex_destroy(&sptd->spt_lock);
        if ((sptcargs->flags & SHM_PAGEABLE) == 0)
                kmem_free(ppa, (sizeof (*ppa) * npages));
out2:
        kmem_free(sptd, sizeof (*sptd));
out1:
        if ((sptcargs->flags & SHM_PAGEABLE) == 0)
                anon_swap_restore(npages);
        return (err);
}

/*ARGSUSED*/
void
segspt_free_pages(struct seg *seg, caddr_t addr, size_t len)
{
        struct page     *pp;
        struct spt_data *sptd = (struct spt_data *)seg->s_data;
```

```
618         pgcnt_t         npages;
619         ulong_t         anon_idx;
620         struct anon_map *amp;
621         struct anon     *ap;
622         struct vnode    *vp;
623         u_offset_t      off;
624         uint_t          hat_flags;
625         int             root = 0;
626         pgcnt_t         pgs, curnpgs = 0;
627         page_t          *rootpp;
628         rctl_qty_t      unlocked_bytes = 0;
629         kproject_t      *proj;
630         kshmid_t        *sp;

632         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
632         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

634         len = P2ROUNDUP(len, PAGESIZE);

636         npages = btop(len);

638         hat_flags = HAT_UNLOAD_UNLOCK | HAT_UNLOAD_UNMAP;
639         if ((hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) ||
640             (sptd->spt_flags & SHM_PAGEABLE)) {
641                 hat_flags = HAT_UNLOAD_UNMAP;
642         }

644         hat_unload(seg->s_as->a_hat, addr, len, hat_flags);

646         amp = sptd->spt_amp;
647         if (sptd->spt_flags & SHM_PAGEABLE)
648                 npages = btop(amp->size);

650         ASSERT(amp != NULL);

652         if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
653                 sp = amp->a_sp;
654                 proj = sp->shm_perm.ipc_proj;
655                 mutex_enter(&sp->shm_mlock);
656         }
657         for (anon_idx = 0; anon_idx < npages; anon_idx++) {
658                 if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
659                         if ((ap = anon_get_ptr(amp->ahp, anon_idx)) == NULL) {
660                                 panic("segspt_free_pages: null app");
661                                 /*NOTREACHED*/
662                         }
663                 } else {
664                         if ((ap = anon_get_next_ptr(amp->ahp, &anon_idx))
665                             == NULL)
666                                 continue;
667                 }
668                 ASSERT(ANON_ISBUSY(anon_get_slot(amp->ahp, anon_idx)) == 0);
669                 swap_xlate(ap, &vp, &off);

671                 /*
672                  * If this platform supports HAT_DYNAMIC_ISM_UNMAP,
673                  * the pages won't be having SE_SHARED lock at this
674                  * point.
675                  *
676                  * On platforms that do not support HAT_DYNAMIC_ISM_UNMAP,
677                  * the pages are still held SE_SHARED locked from the
678                  * original segspt_create()
679                  *
680                  * Our goal is to get SE_EXCL lock on each page, remove
681                  * permanent lock on it and invalidate the page.
682                  */
```



```
683                 if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
684                         if (hat_flags == HAT_UNLOAD_UNMAP)
685                                 pp = page_lookup(vp, off, SE_EXCL);
686                         else {
687                                 if ((pp = page_find(vp, off)) == NULL) {
688                                         panic("segspt_free_pages: "
689                                             "page not locked");
690                                         /*NOTREACHED*/
691                                 }
692                                 if (!page_tryupgrade(pp)) {
693                                         page_unlock(pp);
694                                         pp = page_lookup(vp, off, SE_EXCL);
695                                 }
696                         }
697                         if (pp == NULL) {
698                                 panic("segspt_free_pages: "
699                                     "page not in the system");
700                                 /*NOTREACHED*/
701                         }
702                         ASSERT(pp->p_lckcnt > 0);
703                         page_pp_unlock(pp, 0, 1);
704                         if (pp->p_lckcnt == 0)
705                                 unlocked_bytes += PAGESIZE;
706                 } else {
707                         if ((pp = page_lookup(vp, off, SE_EXCL)) == NULL)
708                                 continue;
709                 }
710                 /*
711                  * It's logical to invalidate the pages here as in most cases
712                  * these were created by segspt.
713                  */
714                 if (pp->p_szc != 0) {
715                         if (root == 0) {
716                                 ASSERT(curnpgs == 0);
717                                 root = 1;
718                                 rootpp = pp;
719                                 pgs = curnpgs = page_get_pagecnt(pp->p_szc);
720                                 ASSERT(pgs > 1);
721                                 ASSERT(IS_P2ALIGNED(pgs, pgs));
722                                 ASSERT(!(page_pptonum(pp) & (pgs - 1)));
723                                 curnpgs--;
724                         } else if ((page_pptonum(pp) & (pgs - 1)) == pgs - 1) {
725                                 ASSERT(curnpgs == 1);
726                                 ASSERT(page_pptonum(pp) ==
727                                     page_pptonum(rootpp) + (pgs - 1));
728                                 page_destroy_pages(rootpp);
729                                 root = 0;
730                                 curnpgs = 0;
731                         } else {
732                                 ASSERT(curnpgs > 1);
733                                 ASSERT(page_pptonum(pp) ==
734                                     page_pptonum(rootpp) + (pgs - curnpgs));
735                                 curnpgs--;
736                         }
737                 } else {
738                         if (root != 0 || curnpgs != 0) {
739                                 panic("segspt_free_pages: bad large page");
740                                 /*NOTREACHED*/
741                         }
742                         /*
743                          * Before destroying the pages, we need to take care
744                          * of the rctl locked memory accounting. For that
745                          * we need to calculte the unlocked_bytes.
746                          */
747                         if (pp->p_lckcnt > 0)
748                                 unlocked_bytes += PAGESIZE;
```

```
749                         /*LINTED: constant in conditional context */
750                         VN_DISPOSE(pp, B_INVAL, 0, kcred);
751                 }
752         }
753         if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
754                 if (unlocked_bytes > 0)
755                         rctl_decr_locked_mem(NULL, proj, unlocked_bytes, 0);
756                 mutex_exit(&sp->shm_mlock);
757         }
758         if (root != 0 || curnpgs != 0) {
759                 panic("segspt_free_pages: bad large page");
760                 /*NOTREACHED*/
761         }

763         /*
764          * mark that pages have been released
765          */
766         sptd->spt_realsize = 0;

768         if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
769                 atomic_add_long(&spt_used, -npages);
770                 anon_swap_restore(npages);
771         }
772 }
```

*unchanged_portion_omitted_*

```
812 /*
813  * DISM only.
814  * Return locked pages over a given range.
815  *
816  * We will cache all DISM locked pages and save the pplist for the
817  * entire segment in the ppa field of the underlying DISM segment structure.
818  * Later, during a call to segspt_reclaim() we will use this ppa array
819  * to page_unlock() all of the pages and then we will free this ppa list.
820  */
821 /*ARGSUSED*/
822 static int
823 segspt_dismpagelock(struct seg *seg, caddr_t addr, size_t len,
824     struct page ***ppp, enum lock_type type, enum seg_rw rw)
825 {
826         struct  shm_data *shmd = (struct shm_data *)seg->s_data;
827         struct  seg     *sptseg = shmd->shm_sptseg;
828         struct  spt_data *sptd = sptseg->s_data;
829         pgcnt_t pg_idx, npages, tot_npages, npgs;
830         struct  page **pplist, **pl, **ppa, *pp;
831         struct  anon_map *amp;
832         spgcnt_t        an_idx;
833         int     ret = ENOTSUP;
834         uint_t  pl_built = 0;
835         struct  anon *ap;
836         struct  vnode *vp;
837         u_offset_t off;
838         pgcnt_t claim_availrmem = 0;
839         uint_t  szc;

841         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
841         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
842         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

844         /*
845          * We want to lock/unlock the entire ISM segment. Therefore,
846          * we will be using the underlying sptseg and it's base address
847          * and length for the caching arguments.
848          */
849         ASSERT(sptseg);
850         ASSERT(sptd);
```



```
852         pg_idx = seg_page(seg, addr);
853         npages = btopr(len);

855         /*
856          * check if the request is larger than number of pages covered
857          * by amp
858          */
859         if (pg_idx + npages > btopr(sptd->spt_amp->size)) {
860                 *ppp = NULL;
861                 return (ENOTSUP);
862         }

864         if (type == L_PAGEUNLOCK) {
865                 ASSERT(sptd->spt_ppa != NULL);

867                 seg_pinactive(seg, NULL, seg->s_base, sptd->spt_amp->size,
868                     sptd->spt_ppa, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);

870                 /*
871                  * If someone is blocked while unmapping, we purge
872                  * segment page cache and thus reclaim pplist synchronously
873                  * without waiting for seg_pasync_thread. This speeds up
874                  * unmapping in cases where munmap(2) is called, while
875                  * raw async i/o is still in progress or where a thread
876                  * exits on data fault in a multithreaded application.
877                  */
878                 if ((sptd->spt_flags & DISM_PPA_CHANGED) ||
879                     (AS_ISUNMAPWAIT(seg->s_as) &&
880                     shmd->shm_softlockcnt > 0)) {
881                         segspt_purge(seg);
882                 }
883                 return (0);
884         }

886         /* The L_PAGELOCK case ... */

888         if (sptd->spt_flags & DISM_PPA_CHANGED) {
889                 segspt_purge(seg);
890                 /*
891                  * for DISM ppa needs to be rebuild since
892                  * number of locked pages could be changed
893                  */
894                 *ppp = NULL;
895                 return (ENOTSUP);
896         }

898         /*
899          * First try to find pages in segment page cache, without
900          * holding the segment lock.
901          */
902         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
903             S_WRITE, SEGP_FORCE_WIRED);
904         if (pplist != NULL) {
905                 ASSERT(sptd->spt_ppa != NULL);
906                 ASSERT(sptd->spt_ppa == pplist);
907                 ppa = sptd->spt_ppa;
908                 for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
909                         if (ppa[an_idx] == NULL) {
910                                 seg_pinactive(seg, NULL, seg->s_base,
911                                     sptd->spt_amp->size, ppa,
912                                     S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
913                                 *ppp = NULL;
914                                 return (ENOTSUP);
915                         }
916                         if ((szc = ppa[an_idx]->p_szc) != 0) {
```

```
917                                 npgs = page_get_pagecnt(szc);
918                                 an_idx = P2ROUNDUP(an_idx + 1, npgs);
919                         } else {
920                                 an_idx++;
921                         }
922                 }
923                 /*
924                  * Since we cache the entire DISM segment, we want to
925                  * set ppp to point to the first slot that corresponds
926                  * to the requested addr, i.e. pg_idx.
927                  */
928                 *ppp = &(sptd->spt_ppa[pg_idx]);
929                 return (0);
930         }

932         mutex_enter(&sptd->spt_lock);
933         /*
934          * try to find pages in segment page cache with mutex
935          */
936         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
937             S_WRITE, SEGP_FORCE_WIRED);
938         if (pplist != NULL) {
939                 ASSERT(sptd->spt_ppa != NULL);
940                 ASSERT(sptd->spt_ppa == pplist);
941                 ppa = sptd->spt_ppa;
942                 for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
943                         if (ppa[an_idx] == NULL) {
944                                 mutex_exit(&sptd->spt_lock);
945                                 seg_pinactive(seg, NULL, seg->s_base,
946                                     sptd->spt_amp->size, ppa,
947                                     S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
948                                 *ppp = NULL;
949                                 return (ENOTSUP);
950                         }
951                         if ((szc = ppa[an_idx]->p_szc) != 0) {
952                                 npgs = page_get_pagecnt(szc);
953                                 an_idx = P2ROUNDUP(an_idx + 1, npgs);
954                         } else {
955                                 an_idx++;
956                         }
957                 }
958                 /*
959                  * Since we cache the entire DISM segment, we want to
960                  * set ppp to point to the first slot that corresponds
961                  * to the requested addr, i.e. pg_idx.
962                  */
963                 mutex_exit(&sptd->spt_lock);
964                 *ppp = &(sptd->spt_ppa[pg_idx]);
965                 return (0);
966         }
967         if (seg_pinsert_check(seg, NULL, seg->s_base, sptd->spt_amp->size,
968             SEGP_FORCE_WIRED) == SEGP_FAIL) {
969                 mutex_exit(&sptd->spt_lock);
970                 *ppp = NULL;
971                 return (ENOTSUP);
972         }

974         /*
975          * No need to worry about protections because DISM pages are always rw.
976          */
977         pl = pplist = NULL;
978         amp = sptd->spt_amp;

980         /*
981          * Do we need to build the ppa array?
982          */
```



```
983         if (sptd->spt_ppa == NULL) {
984                 pgcnt_t lpg_cnt = 0;

986                 pl_built = 1;
987                 tot_npages = btopr(sptd->spt_amp->size);

989                 ASSERT(sptd->spt_pcachecnt == 0);
990                 pplist = kmem_zalloc(sizeof (page_t *) * tot_npages, KM_SLEEP);
991                 pl = pplist;

993                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
994                 for (an_idx = 0; an_idx < tot_npages; ) {
995                         ap = anon_get_ptr(amp->ahp, an_idx);
996                         /*
997                          * Cache only mlocked pages. For large pages
998                          * if one (constituent) page is mlocked
999                          * all pages for that large page
1000                         * are cached also. This is for quick
1001                         * lookups of ppa array;
1002                         */
1003                        if ((ap != NULL) && (lpg_cnt != 0 ||
1004                            (sptd->spt_ppa_lckcnt[an_idx] != 0))) {

1006                                swap_xlate(ap, &vp, &off);
1007                                pp = page_lookup(vp, off, SE_SHARED);
1008                                ASSERT(pp != NULL);
1009                                if (lpg_cnt == 0) {
1010                                        lpg_cnt++;
1011                                        /*
1012                                         * For a small page, we are done --
1013                                         * lpg_count is reset to 0 below.
1014                                         *
1015                                         * For a large page, we are guaranteed
1016                                         * to find the anon structures of all
1017                                         * constituent pages and a non-zero
1018                                         * lpg_cnt ensures that we don't test
1019                                         * for mlock for these. We are done
1020                                         * when lpg_count reaches (npgs + 1).
1021                                         * If we are not the first constituent
1022                                         * page, restart at the first one.
1023                                         */
1024                                        npgs = page_get_pagecnt(pp->p_szc);
1025                                        if (!IS_P2ALIGNED(an_idx, npgs)) {
1026                                                an_idx = P2ALIGN(an_idx, npgs);
1027                                                page_unlock(pp);
1028                                                continue;
1029                                        }
1030                                }
1031                                if (++lpg_cnt > npgs)
1032                                        lpg_cnt = 0;

1034                                /*
1035                                 * availrmem is decremented only
1036                                 * for unlocked pages
1037                                 */
1038                                if (sptd->spt_ppa_lckcnt[an_idx] == 0)
1039                                        claim_availrmem++;
1040                                pplist[an_idx] = pp;
1041                        }
1042                        an_idx++;
1043                }
1044                ANON_LOCK_EXIT(&amp->a_rwlock);

1046                if (claim_availrmem) {
1047                        mutex_enter(&freemem_lock);
1048                        if (availrmem < tune.t_minarmem + claim_availrmem) {
```

```
1049                                mutex_exit(&freemem_lock);
1050                                ret = ENOTSUP;
1051                                claim_availrmem = 0;
1052                                goto insert_fail;
1053                        } else {
1054                                availrmem -= claim_availrmem;
1055                        }
1056                        mutex_exit(&freemem_lock);
1057                }

1059                sptd->spt_ppa = pl;
1060        } else {
1061                /*
1062                 * We already have a valid ppa[].
1063                 */
1064                pl = sptd->spt_ppa;
1065        }

1067        ASSERT(pl != NULL);

1069        ret = seg_pinsert(seg, NULL, seg->s_base, sptd->spt_amp->size,
1070            sptd->spt_amp->size, pl, S_WRITE, SEGP_FORCE_WIRED,
1071            segspt_reclaim);
1072        if (ret == SEGP_FAIL) {
1073                /*
1074                 * seg_pinsert failed. We return
1075                 * ENOTSUP, so that the as_pagelock() code will
1076                 * then try the slower F_SOFTLOCK path.
1077                 */
1078                if (pl_built) {
1079                        /*
1080                         * No one else has referenced the ppa[].
1081                         * We created it and we need to destroy it.
1082                         */
1083                        sptd->spt_ppa = NULL;
1084                }
1085                ret = ENOTSUP;
1086                goto insert_fail;
1087        }

1089        /*
1090         * In either case, we increment softlockcnt on the 'real' segment.
1091         */
1092        sptd->spt_pcachecnt++;
1093        atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));

1095        ppa = sptd->spt_ppa;
1096        for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
1097                if (ppa[an_idx] == NULL) {
1098                        mutex_exit(&sptd->spt_lock);
1099                        seg_pinactive(seg, NULL, seg->s_base,
1100                            sptd->spt_amp->size,
1101                            pl, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
1102                        *ppp = NULL;
1103                        return (ENOTSUP);
1104                }
1105                if ((szc = ppa[an_idx]->p_szc) != 0) {
1106                        npgs = page_get_pagecnt(szc);
1107                        an_idx = P2ROUNDUP(an_idx + 1, npgs);
1108                } else {
1109                        an_idx++;
1110                }
1111        }
1112        /*
1113         * We can now drop the sptd->spt_lock since the ppa[]
1114         * exists and he have incremented pacachecnt.
```



```
1115         */
1116        mutex_exit(&sptd->spt_lock);

1118        /*
1119         * Since we cache the entire segment, we want to
1120         * set ppp to point to the first slot that corresponds
1121         * to the requested addr, i.e. pg_idx.
1122         */
1123        *ppp = &(sptd->spt_ppa[pg_idx]);
1124        return (0);

1126 insert_fail:
1127        /*
1128         * We will only reach this code if we tried and failed.
1129         *
1130         * And we can drop the lock on the dummy seg, once we've failed
1131         * to set up a new ppa[].
1132         */
1133        mutex_exit(&sptd->spt_lock);

1135        if (pl_built) {
1136                if (claim_availrmem) {
1137                        mutex_enter(&freemem_lock);
1138                        availrmem += claim_availrmem;
1139                        mutex_exit(&freemem_lock);
1140                }

1142                /*
1143                 * We created pl and we need to destroy it.
1144                 */
1145                pplist = pl;
1146                for (an_idx = 0; an_idx < tot_npages; an_idx++) {
1147                        if (pplist[an_idx] != NULL)
1148                                page_unlock(pplist[an_idx]);
1149                }
1150                kmem_free(pl, sizeof (page_t *) * tot_npages);
1151        }

1153        if (shmd->shm_softlockcnt <= 0) {
1154                if (AS_ISUNMAPWAIT(seg->s_as)) {
1155                        mutex_enter(&seg->s_as->a_contents);
1156                        if (AS_ISUNMAPWAIT(seg->s_as)) {
1157                                AS_CLRUNMAPWAIT(seg->s_as);
1158                                cv_broadcast(&seg->s_as->a_cv);
1159                        }
1160                        mutex_exit(&seg->s_as->a_contents);
1161                }
1162        }
1163        *ppp = NULL;
1164        return (ret);
1165 }



1169 /*
1170  * return locked pages over a given range.
1171  *
1172  * We will cache the entire ISM segment and save the pplist for the
1173  * entire segment in the ppa field of the underlying ISM segment structure.
1174  * Later, during a call to segspt_reclaim() we will use this ppa array
1175  * to page_unlock() all of the pages and then we will free this ppa list.
1176  */
1177 /*ARGSUSED*/
1178 static int
1179 segspt_shmpagelock(struct seg *seg, caddr_t addr, size_t len,
1180     struct page ***ppp, enum lock_type type, enum seg_rw rw)
```

```
1181 {
1182         struct shm_data *shmd = (struct shm_data *)seg->s_data;
1183         struct seg      *sptseg = shmd->shm_sptseg;
1184         struct spt_data *sptd = sptseg->s_data;
1185         pgcnt_t np, page_index, npages;
1186         caddr_t a, spt_base;
1187         struct page **pplist, **pl, *pp;
1188         struct anon_map *amp;
1189         ulong_t anon_index;
1190         int ret = ENOTSUP;
1191         uint_t  pl_built = 0;
1192         struct anon *ap;
1193         struct vnode *vp;
1194         u_offset_t off;

1196         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
1196         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
1197         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);


1200         /*
1201          * We want to lock/unlock the entire ISM segment. Therefore,
1202          * we will be using the underlying sptseg and it's base address
1203          * and length for the caching arguments.
1204          */
1205         ASSERT(sptseg);
1206         ASSERT(sptd);

1208         if (sptd->spt_flags & SHM_PAGEABLE) {
1209                 return (segspt_dismpagelock(seg, addr, len, ppp, type, rw));
1210         }

1212         page_index = seg_page(seg, addr);
1213         npages = btopr(len);

1215         /*
1216          * check if the request is larger than number of pages covered
1217          * by amp
1218          */
1219         if (page_index + npages > btopr(sptd->spt_amp->size)) {
1220                 *ppp = NULL;
1221                 return (ENOTSUP);
1222         }

1224         if (type == L_PAGEUNLOCK) {

1226                 ASSERT(sptd->spt_ppa != NULL);

1228                 seg_pinactive(seg, NULL, seg->s_base, sptd->spt_amp->size,
1229                     sptd->spt_ppa, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);

1231                 /*
1232                  * If someone is blocked while unmapping, we purge
1233                  * segment page cache and thus reclaim pplist synchronously
1234                  * without waiting for seg_pasync_thread. This speeds up
1235                  * unmapping in cases where munmap(2) is called, while
1236                  * raw async i/o is still in progress or where a thread
1237                  * exits on data fault in a multithreaded application.
1238                  */
1239                 if (AS_ISUNMAPWAIT(seg->s_as) && (shmd->shm_softlockcnt > 0)) {
1240                         segspt_purge(seg);
1241                 }
1242                 return (0);
1243         }

1245         /* The L_PAGELOCK case... */
```



```
1247         /*
1248          * First try to find pages in segment page cache, without
1249          * holding the segment lock.
1250          */
1251         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
1252             S_WRITE, SEGP_FORCE_WIRED);
1253         if (pplist != NULL) {
1254                 ASSERT(sptd->spt_ppa == pplist);
1255                 ASSERT(sptd->spt_ppa[page_index]);
1256                 /*
1257                  * Since we cache the entire ISM segment, we want to
1258                  * set ppp to point to the first slot that corresponds
1259                  * to the requested addr, i.e. page_index.
1260                  */
1261                 *ppp = &(sptd->spt_ppa[page_index]);
1262                 return (0);
1263         }

1265         mutex_enter(&sptd->spt_lock);

1267         /*
1268          * try to find pages in segment page cache
1269          */
1270         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
1271             S_WRITE, SEGP_FORCE_WIRED);
1272         if (pplist != NULL) {
1273                 ASSERT(sptd->spt_ppa == pplist);
1274                 /*
1275                  * Since we cache the entire segment, we want to
1276                  * set ppp to point to the first slot that corresponds
1277                  * to the requested addr, i.e. page_index.
1278                  */
1279                 mutex_exit(&sptd->spt_lock);
1280                 *ppp = &(sptd->spt_ppa[page_index]);
1281                 return (0);
1282         }

1284         if (seg_pinsert_check(seg, NULL, seg->s_base, sptd->spt_amp->size,
1285             SEGP_FORCE_WIRED) == SEGP_FAIL) {
1286                 mutex_exit(&sptd->spt_lock);
1287                 *ppp = NULL;
1288                 return (ENOTSUP);
1289         }

1291         /*
1292          * No need to worry about protections because ISM pages
1293          * are always rw.
1294          */
1295         pl = pplist = NULL;

1297         /*
1298          * Do we need to build the ppa array?
1299          */
1300         if (sptd->spt_ppa == NULL) {
1301                 ASSERT(sptd->spt_ppa == pplist);

1303                 spt_base = sptseg->s_base;
1304                 pl_built = 1;

1306                 /*
1307                  * availrmem is decremented once during anon_swap_adjust()
1308                  * and is incremented during the anon_unresv(), which is
1309                  * called from shm_rm_amp() when the segment is destroyed.
1310                  */
1311                 amp = sptd->spt_amp;
```

```
1312                ASSERT(amp != NULL);

1314                /* pcachecnt is protected by sptd->spt_lock */
1315                ASSERT(sptd->spt_pcachecnt == 0);
1316                pplist = kmem_zalloc(sizeof (page_t *)
1317                    * btopr(sptd->spt_amp->size), KM_SLEEP);
1318                pl = pplist;

1320                anon_index = seg_page(sptseg, spt_base);

1322                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
1323                for (a = spt_base; a < (spt_base + sptd->spt_amp->size);
1324                    a += PAGESIZE, anon_index++, pplist++) {
1325                        ap = anon_get_ptr(amp->ahp, anon_index);
1326                        ASSERT(ap != NULL);
1327                        swap_xlate(ap, &vp, &off);
1328                        pp = page_lookup(vp, off, SE_SHARED);
1329                        ASSERT(pp != NULL);
1330                        *pplist = pp;
1331                }
1332                ANON_LOCK_EXIT(&amp->a_rwlock);

1334                if (a < (spt_base + sptd->spt_amp->size)) {
1335                        ret = ENOTSUP;
1336                        goto insert_fail;
1337                }
1338                sptd->spt_ppa = pl;
1339        } else {
1340                /*
1341                 * We already have a valid ppa[].
1342                 */
1343                pl = sptd->spt_ppa;
1344        }

1346        ASSERT(pl != NULL);

1348        ret = seg_pinsert(seg, NULL, seg->s_base, sptd->spt_amp->size,
1349            sptd->spt_amp->size, pl, S_WRITE, SEGP_FORCE_WIRED,
1350            segspt_reclaim);
1351        if (ret == SEGP_FAIL) {
1352                /*
1353                 * seg_pinsert failed. We return
1354                 * ENOTSUP, so that the as_pagelock() code will
1355                 * then try the slower F_SOFTLOCK path.
1356                 */
1357                if (pl_built) {
1358                        /*
1359                         * No one else has referenced the ppa[].
1360                         * We created it and we need to destroy it.
1361                         */
1362                        sptd->spt_ppa = NULL;
1363                }
1364                ret = ENOTSUP;
1365                goto insert_fail;
1366        }

1368        /*
1369         * In either case, we increment softlockcnt on the 'real' segment.
1370         */
1371        sptd->spt_pcachecnt++;
1372        atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));

1374        /*
1375         * We can now drop the sptd->spt_lock since the ppa[]
1376         * exists and he have incremented pacachecnt.
1377         */
```



```
1378        mutex_exit(&sptd->spt_lock);

1380        /*
1381         * Since we cache the entire segment, we want to
1382         * set ppp to point to the first slot that corresponds
1383         * to the requested addr, i.e. page_index.
1384         */
1385        *ppp = &(sptd->spt_ppa[page_index]);
1386        return (0);

1388 insert_fail:
1389        /*
1390         * We will only reach this code if we tried and failed.
1391         *
1392         * And we can drop the lock on the dummy seg, once we've failed
1393         * to set up a new ppa[].
1394         */
1395        mutex_exit(&sptd->spt_lock);

1397        if (pl_built) {
1398                /*
1399                 * We created pl and we need to destroy it.
1400                 */
1401                pplist = pl;
1402                np = (((uintptr_t)(a - spt_base)) >> PAGESHIFT);
1403                while (np) {
1404                        page_unlock(*pplist);
1405                        np--;
1406                        pplist++;
1407                }
1408                kmem_free(pl, sizeof (page_t *) * btopr(sptd->spt_amp->size));
1409        }
1410        if (shmd->shm_softlockcnt <= 0) {
1411                if (AS_ISUNMAPWAIT(seg->s_as)) {
1412                        mutex_enter(&seg->s_as->a_contents);
1413                        if (AS_ISUNMAPWAIT(seg->s_as)) {
1414                                AS_CLRUNMAPWAIT(seg->s_as);
1415                                cv_broadcast(&seg->s_as->a_cv);
1416                        }
1417                        mutex_exit(&seg->s_as->a_contents);
1418                }
1419        }
1420        *ppp = NULL;
1421        return (ret);
1422 }
```

_____***unchanged_portion_omitted_***

```
1433 static int
1434 segspt_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
1435        enum seg_rw rw, int async)
1436 {
1437        struct seg *seg = (struct seg *)ptag;
1438        struct  shm_data *shmd = (struct shm_data *)seg->s_data;
1439        struct  seg     *sptseg;
1440        struct  spt_data *sptd;
1441        pgcnt_t npages, i, free_availrmem = 0;
1442        int     done = 0;

1444 #ifdef lint
1445        addr = addr;
1446 #endif
1447        sptseg = shmd->shm_sptseg;
1448        sptd = sptseg->s_data;
1449        npages = (len >> PAGESHIFT);
1450        ASSERT(npages);
1451        ASSERT(sptd->spt_pcachecnt != 0);
```

```
1452         ASSERT(sptd->spt_ppa == pplist);
1453         ASSERT(npages == btopr(sptd->spt_amp->size));
1454         ASSERT(async || AS_LOCK_HELD(seg->s_as));
1454         ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1456         /*
1457          * Acquire the lock on the dummy seg and destroy the
1458          * ppa array IF this is the last pcachecnt.
1459          */
1460         mutex_enter(&sptd->spt_lock);
1461         if (--sptd->spt_pcachecnt == 0) {
1462                 for (i = 0; i < npages; i++) {
1463                         if (pplist[i] == NULL) {
1464                                 continue;
1465                         }
1466                         if (rw == S_WRITE) {
1467                                 hat_setrefmod(pplist[i]);
1468                         } else {
1469                                 hat_setref(pplist[i]);
1470                         }
1471                         if ((sptd->spt_flags & SHM_PAGEABLE) &&
1472                             (sptd->spt_ppa_lckcnt[i] == 0))
1473                                 free_availrmem++;
1474                         page_unlock(pplist[i]);
1475                 }
1476                 if ((sptd->spt_flags & SHM_PAGEABLE) && free_availrmem) {
1477                         mutex_enter(&freemem_lock);
1478                         availrmem += free_availrmem;
1479                         mutex_exit(&freemem_lock);
1480                 }
1481                 /*
1482                  * Since we want to cach/uncache the entire ISM segment,
1483                  * we will track the pplist in a segspt specific field
1484                  * ppa, that is initialized at the time we add an entry to
1485                  * the cache.
1486                  */
1487                 ASSERT(sptd->spt_pcachecnt == 0);
1488                 kmem_free(pplist, sizeof (page_t *) * npages);
1489                 sptd->spt_ppa = NULL;
1490                 sptd->spt_flags &= ~DISM_PPA_CHANGED;
1491                 sptd->spt_gen++;
1492                 cv_broadcast(&sptd->spt_cv);
1493                 done = 1;
1494         }
1495         mutex_exit(&sptd->spt_lock);

1497         /*
1498          * If we are pcache async thread or called via seg_ppurge_wiredpp() we
1499          * may not hold AS lock (in this case async argument is not 0). This
1500          * means if softlockcnt drops to 0 after the decrement below address
1501          * space may get freed. We can't allow it since after softlock
1502          * derement to 0 we still need to access as structure for possible
1503          * wakeup of unmap waiters. To prevent the disappearance of as we take
1504          * this segment's shm_segfree_syncmtx. segspt_shmfree() also takes
1505          * this mutex as a barrier to make sure this routine completes before
1506          * segment is freed.
1507          *
1508          * The second complication we have to deal with in async case is a
1509          * possibility of missed wake up of unmap wait thread. When we don't
1510          * hold as lock here we may take a_contents lock before unmap wait
1511          * thread that was first to see softlockcnt was still not 0. As a
1512          * result we'll fail to wake up an unmap wait thread. To avoid this
1513          * race we set nounmapwait flag in as structure if we drop softlockcnt
1514          * to 0 if async is not 0.  unmapwait thread
1515          * will not block if this flag is set.
1516          */
```



```
1517         if (async)
1518                 mutex_enter(&shmd->shm_segfree_syncmtx);

1520         /*
1521          * Now decrement softlockcnt.
1522          */
1523         ASSERT(shmd->shm_softlockcnt > 0);
1524         atomic_dec_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));

1526         if (shmd->shm_softlockcnt <= 0) {
1527                 if (async || AS_ISUNMAPWAIT(seg->s_as)) {
1528                         mutex_enter(&seg->s_as->a_contents);
1529                         if (async)
1530                                 AS_SETNOUNMAPWAIT(seg->s_as);
1531                         if (AS_ISUNMAPWAIT(seg->s_as)) {
1532                                 AS_CLRUNMAPWAIT(seg->s_as);
1533                                 cv_broadcast(&seg->s_as->a_cv);
1534                         }
1535                         mutex_exit(&seg->s_as->a_contents);
1536                 }
1537         }

1539         if (async)
1540                 mutex_exit(&shmd->shm_segfree_syncmtx);

1542         return (done);
1543 }

1545 /*
1546  * Do a F_SOFTUNLOCK call over the range requested.
1547  * The range must have already been F_SOFTLOCK'ed.
1548  *
1549  * The calls to acquire and release the anon map lock mutex were
1550  * removed in order to avoid a deadly embrace during a DR
1551  * memory delete operation.  (Eg. DR blocks while waiting for a
1552  * exclusive lock on a page that is being used for kaio; the
1553  * thread that will complete the kaio and call segspt_softunlock
1554  * blocks on the anon map lock; another thread holding the anon
1555  * map lock blocks on another page lock via the segspt_shmfault
1556  * -> page_lookup -> page_lookup_create -> page_lock_es code flow.)
1557  *
1558  * The appropriateness of the removal is based upon the following:
1559  * 1. If we are holding a segment's reader lock and the page is held
1560  * shared, then the corresponding element in anonmap which points to
1561  * anon struct cannot change and there is no need to acquire the
1562  * anonymous map lock.
1563  * 2. Threads in segspt_softunlock have a reader lock on the segment
1564  * and already have the shared page lock, so we are guaranteed that
1565  * the anon map slot cannot change and therefore can call anon_get_ptr()
1566  * without grabbing the anonymous map lock.
1567  * 3. Threads that softlock a shared page break copy-on-write, even if
1568  * its a read.  Thus cow faults can be ignored with respect to soft
1569  * unlocking, since the breaking of cow means that the anon slot(s) will
1570  * not be shared.
1571  */
1572 static void
1573 segspt_softunlock(struct seg *seg, caddr_t sptseg_addr,
1574         size_t len, enum seg_rw rw)
1575 {
1576         struct shm_data *shmd = (struct shm_data *)seg->s_data;
1577         struct seg      *sptseg;
1578         struct spt_data *sptd;
1579         page_t *pp;
1580         caddr_t adr;
1581         struct vnode *vp;
1582         u_offset_t offset;
```

```
1583         ulong_t anon_index;
1584         struct anon_map *amp;           /* XXX - for locknest */
1585         struct anon *ap = NULL;
1586         pgcnt_t npages;

1588         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
1588         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1590         sptseg = shmd->shm_sptseg;
1591         sptd = sptseg->s_data;

1593         /*
1594          * Some platforms assume that ISM mappings are HAT_LOAD_LOCK
1595          * and therefore their pages are SE_SHARED locked
1596          * for the entire life of the segment.
1597          */
1598         if ((!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) &&
1599             ((sptd->spt_flags & SHM_PAGEABLE) == 0)) {
1600                 goto softlock_decrement;
1601         }

1603         /*
1604          * Any thread is free to do a page_find and
1605          * page_unlock() on the pages within this seg.
1606          *
1607          * We are already holding the as->a_lock on the user's
1608          * real segment, but we need to hold the a_lock on the
1609          * underlying dummy as. This is mostly to satisfy the
1610          * underlying HAT layer.
1611          */
1612         AS_LOCK_ENTER(sptseg->s_as, RW_READER);
1612         AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
1613         hat_unlock(sptseg->s_as->a_hat, sptseg_addr, len);
1614         AS_LOCK_EXIT(sptseg->s_as);
1614         AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);

1616         amp = sptd->spt_amp;
1617         ASSERT(amp != NULL);
1618         anon_index = seg_page(sptseg, sptseg_addr);

1620         for (adr = sptseg_addr; adr < sptseg_addr + len; adr += PAGESIZE) {
1621                 ap = anon_get_ptr(amp->ahp, anon_index++);
1622                 ASSERT(ap != NULL);
1623                 swap_xlate(ap, &vp, &offset);

1625                 /*
1626                  * Use page_find() instead of page_lookup() to
1627                  * find the page since we know that it has a
1628                  * "shared" lock.
1629                  */
1630                 pp = page_find(vp, offset);
1631                 ASSERT(ap == anon_get_ptr(amp->ahp, anon_index - 1));
1632                 if (pp == NULL) {
1633                         panic("segspt_softunlock: "
1634                             "addr %p, ap %p, vp %p, off %llx",
1635                             (void *)adr, (void *)ap, (void *)vp, offset);
1636                         /*NOTREACHED*/
1637                 }

1639                 if (rw == S_WRITE) {
1640                         hat_setrefmod(pp);
1641                 } else if (rw != S_OTHER) {
1642                         hat_setref(pp);
1643                 }
1644                 page_unlock(pp);
1645         }
```



```
1647 softlock_decrement:
1648         npages = btopr(len);
1649         ASSERT(shmd->shm_softlockcnt >= npages);
1650         atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), -npages);
1651         if (shmd->shm_softlockcnt == 0) {
1652                 /*
1653                  * All SOFTLOCKS are gone. Wakeup any waiting
1654                  * unmappers so they can try again to unmap.
1655                  * Check for waiters first without the mutex
1656                  * held so we don't always grab the mutex on
1657                  * softunlocks.
1658                  */
1659                 if (AS_ISUNMAPWAIT(seg->s_as)) {
1660                         mutex_enter(&seg->s_as->a_contents);
1661                         if (AS_ISUNMAPWAIT(seg->s_as)) {
1662                                 AS_CLRUNMAPWAIT(seg->s_as);
1663                                 cv_broadcast(&seg->s_as->a_cv);
1664                         }
1665                         mutex_exit(&seg->s_as->a_contents);
1666                 }
1667         }
1668 }

1670 int
1671 segspt_shmattach(struct seg *seg, caddr_t *argsp)
1672 {
1673         struct shm_data *shmd_arg = (struct shm_data *)argsp;
1674         struct shm_data *shmd;
1675         struct anon_map *shm_amp = shmd_arg->shm_amp;
1676         struct spt_data *sptd;
1677         int error = 0;

1679         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1679         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1681         shmd = kmem_zalloc((sizeof (*shmd)), KM_NOSLEEP);
1682         if (shmd == NULL)
1683                 return (ENOMEM);

1685         shmd->shm_sptas = shmd_arg->shm_sptas;
1686         shmd->shm_amp = shm_amp;
1687         shmd->shm_sptseg = shmd_arg->shm_sptseg;

1689         (void) lgrp_shm_policy_set(LGRP_MEM_POLICY_DEFAULT, shm_amp, 0,
1690             NULL, 0, seg->s_size);

1692         mutex_init(&shmd->shm_segfree_syncmtx, NULL, MUTEX_DEFAULT, NULL);

1694         seg->s_data = (void *)shmd;
1695         seg->s_ops = &segspt_shmops;
1696         seg->s_szc = shmd->shm_sptseg->s_szc;
1697         sptd = shmd->shm_sptseg->s_data;

1699         if (sptd->spt_flags & SHM_PAGEABLE) {
1700                 if ((shmd->shm_vpage = kmem_zalloc(btopr(shm_amp->size),
1701                     KM_NOSLEEP)) == NULL) {
1702                         seg->s_data = (void *)NULL;
1703                         kmem_free(shmd, (sizeof (*shmd)));
1704                         return (ENOMEM);
1705                 }
1706                 shmd->shm_lckpgs = 0;
1707                 if (hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
1708                         if ((error = hat_share(seg->s_as->a_hat, seg->s_base,
1709                             shmd_arg->shm_sptas->a_hat, SEGSPTADDR,
1710                             seg->s_size, seg->s_szc)) != 0) {
```

```
1711                                kmem_free(shmd->shm_vpage,
1712                                    btopr(shm_amp->size));
1713                        }
1714                }
1715        } else {
1716                error = hat_share(seg->s_as->a_hat, seg->s_base,
1717                    shmd_arg->shm_sptas->a_hat, SEGSPTADDR,
1718                    seg->s_size, seg->s_szc);
1719        }
1720        if (error) {
1721                seg->s_szc = 0;
1722                seg->s_data = (void *)NULL;
1723                kmem_free(shmd, (sizeof (*shmd)));
1724        } else {
1725                ANON_LOCK_ENTER(&shm_amp->a_rwlock, RW_WRITER);
1726                shm_amp->refcnt++;
1727                ANON_LOCK_EXIT(&shm_amp->a_rwlock);
1728        }
1729        return (error);
1730 }

1732 int
1733 segspt_shmunmap(struct seg *seg, caddr_t raddr, size_t ssize)
1734 {
1735        struct shm_data *shmd = (struct shm_data *)seg->s_data;
1736        int reclaim = 1;

1738        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1738        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
1739 retry:
1740        if (shmd->shm_softlockcnt > 0) {
1741                if (reclaim == 1) {
1742                        segspt_purge(seg);
1743                        reclaim = 0;
1744                        goto retry;
1745                }
1746                return (EAGAIN);
1747        }

1749        if (ssize != seg->s_size) {
1750 #ifdef DEBUG
1751                cmn_err(CE_WARN, "Incompatible ssize %lx s_size %lx\n",
1752                    ssize, seg->s_size);
1753 #endif
1754                return (EINVAL);
1755        }

1757        (void) segspt_shmlockop(seg, raddr, shmd->shm_amp->size, 0, MC_UNLOCK,
1758            NULL, 0);
1759        hat_unshare(seg->s_as->a_hat, raddr, ssize, seg->s_szc);

1761        seg_free(seg);

1763        return (0);
1764 }

1766 void
1767 segspt_shmfree(struct seg *seg)
1768 {
1769        struct shm_data *shmd = (struct shm_data *)seg->s_data;
1770        struct anon_map *shm_amp = shmd->shm_amp;

1772        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1772        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1774        (void) segspt_shmlockop(seg, seg->s_base, shm_amp->size, 0,
```



```
1775            MC_UNLOCK, NULL, 0);

1777        /*
1778         * Need to increment refcnt when attaching
1779         * and decrement when detaching because of dup().
1780         */
1781        ANON_LOCK_ENTER(&shm_amp->a_rwlock, RW_WRITER);
1782        shm_amp->refcnt--;
1783        ANON_LOCK_EXIT(&shm_amp->a_rwlock);

1785        if (shmd->shm_vpage) {  /* only for DISM */
1786                kmem_free(shmd->shm_vpage, btopr(shm_amp->size));
1787                shmd->shm_vpage = NULL;
1788        }

1790        /*
1791         * Take shm_segfree_syncmtx lock to let segspt_reclaim() finish if it's
1792         * still working with this segment without holding as lock.
1793         */
1794        ASSERT(shmd->shm_softlockcnt == 0);
1795        mutex_enter(&shmd->shm_segfree_syncmtx);
1796        mutex_destroy(&shmd->shm_segfree_syncmtx);

1798        kmem_free(shmd, sizeof (*shmd));
1799 }

1801 /*ARGSUSED*/
1802 int
1803 segspt_shmsetprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
1804 {
1805        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
1805        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1807        /*
1808         * Shared page table is more than shared mapping.
1809         *  Individual process sharing page tables can't change prot
1810         *  because there is only one set of page tables.
1811         *  This will be allowed after private page table is
1812         *  supported.
1813         */
1814 /* need to return correct status error? */
1815        return (0);
1816 }


1819 faultcode_t
1820 segspt_dismfault(struct hat *hat, struct seg *seg, caddr_t addr,
1821     size_t len, enum fault_type type, enum seg_rw rw)
1822 {
1823        struct  shm_data        *shmd = (struct shm_data *)seg->s_data;
1824        struct  seg             *sptseg = shmd->shm_sptseg;
1825        struct  as              *curspt = shmd->shm_sptas;
1826        struct  spt_data        *sptd = sptseg->s_data;
1827        pgcnt_t npages;
1828        size_t  size;
1829        caddr_t segspt_addr, shm_addr;
1830        page_t  **ppa;
1831        int     i;
1832        ulong_t an_idx = 0;
1833        int     err = 0;
1834        int     dyn_ism_unmap = hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0);
1835        size_t  pgsz;
1836        pgcnt_t pgcnt;
1837        caddr_t a;
1838        pgcnt_t pidx;
```

```
1840 #ifdef lint
1841         hat = hat;
1842 #endif
1843         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
1843         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1845         /*
1846          * Because of the way spt is implemented
1847          * the realsize of the segment does not have to be
1848          * equal to the segment size itself. The segment size is
1849          * often in multiples of a page size larger than PAGESIZE.
1850          * The realsize is rounded up to the nearest PAGESIZE
1851          * based on what the user requested. This is a bit of
1852          * ungliness that is historical but not easily fixed
1853          * without re-designing the higher levels of ISM.
1854          */
1855         ASSERT(addr >= seg->s_base);
1856         if (((addr + len) - seg->s_base) > sptd->spt_realsize)
1857               return (FC_NOMAP);
1858         /*
1859          * For all of the following cases except F_PROT, we need to
1860          * make any necessary adjustments to addr and len
1861          * and get all of the necessary page_t's into an array called ppa[].
1862          *
1863          * The code in shmat() forces base addr and len of ISM segment
1864          * to be aligned to largest page size supported. Therefore,
1865          * we are able to handle F_SOFTLOCK and F_INVAL calls in "large
1866          * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
1867          * in large pagesize chunks, or else we will screw up the HAT
1868          * layer by calling hat_memload_array() with differing page sizes
1869          * over a given virtual range.
1870          */
1871         pgsz = page_get_pagesize(sptseg->s_szc);
1872         pgcnt = page_get_pagecnt(sptseg->s_szc);
1873         shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
1874         size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)), pgsz);
1875         npages = btopr(size);

1877         /*
1878          * Now we need to convert from addr in segshm to addr in segspt.
1879          */
1880         an_idx = seg_page(seg, shm_addr);
1881         segspt_addr = sptseg->s_base + ptob(an_idx);

1883         ASSERT((segspt_addr + ptob(npages)) <=
1884             (sptseg->s_base + sptd->spt_realsize));
1885         ASSERT(segspt_addr < (sptseg->s_base + sptseg->s_size));

1887         switch (type) {

1889         case F_SOFTLOCK:

1891                 atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), npages);
1892                 /*
1893                  * Fall through to the F_INVAL case to load up the hat layer
1894                  * entries with the HAT_LOAD_LOCK flag.
1895                  */
1896                 /* FALLTHRU */
1897         case F_INVAL:

1899                 if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
1900                         return (FC_NOMAP);

1902                 ppa = kmem_zalloc(npages * sizeof (page_t *), KM_SLEEP);

1904                 err = spt_anon_getpages(sptseg, segspt_addr, size, ppa);
```



```
1905                 if (err != 0) {
1906                         if (type == F_SOFTLOCK) {
1907                                 atomic_add_long((ulong_t *)(
1908                                     &(shmd->shm_softlockcnt)), -npages);
1909                         }
1910                         goto dism_err;
1911                 }
1912                 AS_LOCK_ENTER(sptseg->s_as, RW_READER);
1912                 AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
1913                 a = segspt_addr;
1914                 pidx = 0;
1915                 if (type == F_SOFTLOCK) {

1917                         /*
1918                          * Load up the translation keeping it
1919                          * locked and don't unlock the page.
1920                          */
1921                         for (; pidx < npages; a += pgsz, pidx += pgcnt) {
1922                                 hat_memload_array(sptseg->s_as->a_hat,
1923                                     a, pgsz, &ppa[pidx], sptd->spt_prot,
1924                                     HAT_LOAD_LOCK | HAT_LOAD_SHARE);
1925                         }
1926                 } else {
1927                         if (hat == seg->s_as->a_hat) {

1929                                 /*
1930                                  * Migrate pages marked for migration
1931                                  */
1932                                 if (lgrp_optimizations())
1933                                         page_migrate(seg, shm_addr, ppa,
1934                                             npages);

1936                                 /* CPU HAT */
1937                                 for (; pidx < npages;
1938                                     a += pgsz, pidx += pgcnt) {
1939                                         hat_memload_array(sptseg->s_as->a_hat,
1940                                             a, pgsz, &ppa[pidx],
1941                                             sptd->spt_prot,
1942                                             HAT_LOAD_SHARE);
1943                                 }
1944                         } else {
1945                                 /* XHAT. Pass real address */
1946                                 hat_memload_array(hat, shm_addr,
1947                                     size, ppa, sptd->spt_prot, HAT_LOAD_SHARE);
1948                         }

1950                         /*
1951                          * And now drop the SE_SHARED lock(s).
1952                          */
1953                         if (dyn_ism_unmap) {
1954                                 for (i = 0; i < npages; i++) {
1955                                         page_unlock(ppa[i]);
1956                                 }
1957                         }
1958                 }

1960                 if (!dyn_ism_unmap) {
1961                         if (hat_share(seg->s_as->a_hat, shm_addr,
1962                             curspt->a_hat, segspt_addr, ptob(npages),
1963                             seg->s_szc) != 0) {
1964                                 panic("hat_share err in DISM fault");
1965                                 /* NOTREACHED */
1966                         }
1967                         if (type == F_INVAL) {
1968                                 for (i = 0; i < npages; i++) {
1969                                         page_unlock(ppa[i]);
```

```
1970                                    }
1971                            }
1972                    }
1973                    AS_LOCK_EXIT(sptseg->s_as);
1973                    AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);
1974 dism_err:
1975                    kmem_free(ppa, npages * sizeof (page_t *));
1976                    return (err);

1978            case F_SOFTUNLOCK:

1980                    /*
1981                     * This is a bit ugly, we pass in the real seg pointer,
1982                     * but the segspt_addr is the virtual address within the
1983                     * dummy seg.
1984                     */
1985                    segspt_softunlock(seg, segspt_addr, size, rw);
1986                    return (0);

1988            case F_PROT:

1990                    /*
1991                     * This takes care of the unusual case where a user
1992                     * allocates a stack in shared memory and a register
1993                     * window overflow is written to that stack page before
1994                     * it is otherwise modified.
1995                     *
1996                     * We can get away with this because ISM segments are
1997                     * always rw. Other than this unusual case, there
1998                     * should be no instances of protection violations.
1999                     */
2000                    return (0);

2002            default:
2003 #ifdef DEBUG
2004                    panic("segspt_dismfault default type?");
2005 #else
2006                    return (FC_NOMAP);
2007 #endif
2008            }
2009 }


2012 faultcode_t
2013 segspt_shmfault(struct hat *hat, struct seg *seg, caddr_t addr,
2014     size_t len, enum fault_type type, enum seg_rw rw)
2015 {
2016            struct shm_data         *shmd = (struct shm_data *)seg->s_data;
2017            struct seg              *sptseg = shmd->shm_sptseg;
2018            struct as               *curspt = shmd->shm_sptas;
2019            struct spt_data         *sptd   = sptseg->s_data;
2020            pgcnt_t npages;
2021            size_t size;
2022            caddr_t sptseg_addr, shm_addr;
2023            page_t *pp, **ppa;
2024            int     i;
2025            u_offset_t offset;
2026            ulong_t anon_index = 0;
2027            struct vnode *vp;
2028            struct anon_map *amp;           /* XXX - for locknest */
2029            struct anon *ap = NULL;
2030            size_t          pgsz;
2031            pgcnt_t         pgcnt;
2032            caddr_t         a;
2033            pgcnt_t         pidx;
2034            size_t          sz;
```



```
2036 #ifdef lint
2037            hat = hat;
2038 #endif

2040            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2040            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2042            if (sptd->spt_flags & SHM_PAGEABLE) {
2043                    return (segspt_dismfault(hat, seg, addr, len, type, rw));
2044            }

2046            /*
2047             * Because of the way spt is implemented
2048             * the realsize of the segment does not have to be
2049             * equal to the segment size itself. The segment size is
2050             * often in multiples of a page size larger than PAGESIZE.
2051             * The realsize is rounded up to the nearest PAGESIZE
2052             * based on what the user requested. This is a bit of
2053             * ungliness that is historical but not easily fixed
2054             * without re-designing the higher levels of ISM.
2055             */
2056            ASSERT(addr >= seg->s_base);
2057            if (((addr + len) - seg->s_base) > sptd->spt_realsize)
2058                    return (FC_NOMAP);
2059            /*
2060             * For all of the following cases except F_PROT, we need to
2061             * make any necessary adjustments to addr and len
2062             * and get all of the necessary page_t's into an array called ppa[].
2063             *
2064             * The code in shmat() forces base addr and len of ISM segment
2065             * to be aligned to largest page size supported. Therefore,
2066             * we are able to handle F_SOFTLOCK and F_INVAL calls in "large
2067             * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
2068             * in large pagesize chunks, or else we will screw up the HAT
2069             * layer by calling hat_memload_array() with differing page sizes
2070             * over a given virtual range.
2071             */
2072            pgsz = page_get_pagesize(sptseg->s_szc);
2073            pgcnt = page_get_pagecnt(sptseg->s_szc);
2074            shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
2075            size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)), pgsz);
2076            npages = btopr(size);

2078            /*
2079             * Now we need to convert from addr in segshm to addr in segspt.
2080             */
2081            anon_index = seg_page(seg, shm_addr);
2082            sptseg_addr = sptseg->s_base + ptob(anon_index);

2084            /*
2085             * And now we may have to adjust npages downward if we have
2086             * exceeded the realsize of the segment or initial anon
2087             * allocations.
2088             */
2089            if ((sptseg_addr + ptob(npages)) >
2090                (sptseg->s_base + sptd->spt_realsize))
2091                    size = (sptseg->s_base + sptd->spt_realsize) - sptseg_addr;

2093            npages = btopr(size);

2095            ASSERT(sptseg_addr < (sptseg->s_base + sptseg->s_size));
2096            ASSERT((sptd->spt_flags & SHM_PAGEABLE) == 0);

2098            switch (type) {
```

```
2100         case F_SOFTLOCK:

2102                 /*
2103                  * availrmem is decremented once during anon_swap_adjust()
2104                  * and is incremented during the anon_unresv(), which is
2105                  * called from shm_rm_amp() when the segment is destroyed.
2106                  */
2107                 atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), npages);
2108                 /*
2109                  * Some platforms assume that ISM pages are SE_SHARED
2110                  * locked for the entire life of the segment.
2111                  */
2112                 if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0))
2113                         return (0);
2114                 /*
2115                  * Fall through to the F_INVAL case to load up the hat layer
2116                  * entries with the HAT_LOAD_LOCK flag.
2117                  */

2119                 /* FALLTHRU */
2120         case F_INVAL:

2122                 if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
2123                         return (FC_NOMAP);

2125                 /*
2126                  * Some platforms that do NOT support DYNAMIC_ISM_UNMAP
2127                  * may still rely on this call to hat_share(). That
2128                  * would imply that those hat's can fault on a
2129                  * HAT_LOAD_LOCK translation, which would seem
2130                  * contradictory.
2131                  */
2132                 if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
2133                         if (hat_share(seg->s_as->a_hat, seg->s_base,
2134                             curspt->a_hat, sptseg->s_base,
2135                             sptseg->s_size, sptseg->s_szc) != 0) {
2136                                 panic("hat_share error in ISM fault");
2137                                 /*NOTREACHED*/
2138                         }
2139                         return (0);
2140                 }
2141                 ppa = kmem_zalloc(sizeof (page_t *) * npages, KM_SLEEP);

2143                 /*
2144                  * I see no need to lock the real seg,
2145                  * here, because all of our work will be on the underlying
2146                  * dummy seg.
2147                  *
2148                  * sptseg_addr and npages now account for large pages.
2149                  */
2150                 amp = sptd->spt_amp;
2151                 ASSERT(amp != NULL);
2152                 anon_index = seg_page(sptseg, sptseg_addr);

2154                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2155                 for (i = 0; i < npages; i++) {
2156                         ap = anon_get_ptr(amp->ahp, anon_index++);
2157                         ASSERT(ap != NULL);
2158                         swap_xlate(ap, &vp, &offset);
2159                         pp = page_lookup(vp, offset, SE_SHARED);
2160                         ASSERT(pp != NULL);
2161                         ppa[i] = pp;
2162                 }
2163                 ANON_LOCK_EXIT(&amp->a_rwlock);
2164                 ASSERT(i == npages);
```



```
2166                 /*
2167                  * We are already holding the as->a_lock on the user's
2168                  * real segment, but we need to hold the a_lock on the
2169                  * underlying dummy as. This is mostly to satisfy the
2170                  * underlying HAT layer.
2171                  */
2172                 AS_LOCK_ENTER(sptseg->s_as, RW_READER);
2172                 AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
2173                 a = sptseg_addr;
2174                 pidx = 0;
2175                 if (type == F_SOFTLOCK) {
2176                         /*
2177                          * Load up the translation keeping it
2178                          * locked and don't unlock the page.
2179                          */
2180                         for (; pidx < npages; a += pgsz, pidx += pgcnt) {
2181                                 sz = MIN(pgsz, ptob(npages - pidx));
2182                                 hat_memload_array(sptseg->s_as->a_hat, a,
2183                                     sz, &ppa[pidx], sptd->spt_prot,
2184                                     HAT_LOAD_LOCK | HAT_LOAD_SHARE);
2185                         }
2186                 } else {
2187                         if (hat == seg->s_as->a_hat) {

2189                                 /*
2190                                  * Migrate pages marked for migration.
2191                                  */
2192                                 if (lgrp_optimizations())
2193                                         page_migrate(seg, shm_addr, ppa,
2194                                             npages);

2196                                 /* CPU HAT */
2197                                 for (; pidx < npages;
2198                                     a += pgsz, pidx += pgcnt) {
2199                                         sz = MIN(pgsz, ptob(npages - pidx));
2200                                         hat_memload_array(sptseg->s_as->a_hat,
2201                                             a, sz, &ppa[pidx],
2202                                             sptd->spt_prot, HAT_LOAD_SHARE);
2203                                 }
2204                         } else {
2205                                 /* XHAT. Pass real address */
2206                                 hat_memload_array(hat, shm_addr,
2207                                     ptob(npages), ppa, sptd->spt_prot,
2208                                     HAT_LOAD_SHARE);
2209                         }

2211                         /*
2212                          * And now drop the SE_SHARED lock(s).
2213                          */
2214                         for (i = 0; i < npages; i++)
2215                                 page_unlock(ppa[i]);
2216                 }
2217                 AS_LOCK_EXIT(sptseg->s_as);
2217                 AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);

2219                 kmem_free(ppa, sizeof (page_t *) * npages);
2220                 return (0);
2221         case F_SOFTUNLOCK:

2223                 /*
2224                  * This is a bit ugly, we pass in the real seg pointer,
2225                  * but the sptseg_addr is the virtual address within the
2226                  * dummy seg.
2227                  */
2228                 segspt_softunlock(seg, sptseg_addr, ptob(npages), rw);
2229                 return (0);
```

```
2231         case F_PROT:

2233                 /*
2234                  * This takes care of the unusual case where a user
2235                  * allocates a stack in shared memory and a register
2236                  * window overflow is written to that stack page before
2237                  * it is otherwise modified.
2238                  *
2239                  * We can get away with this because ISM segments are
2240                  * always rw. Other than this unusual case, there
2241                  * should be no instances of protection violations.
2242                  */
2243                 return (0);

2245         default:
2246 #ifdef DEBUG
2247                 cmn_err(CE_WARN, "segspt_shmfault default type?");
2248 #endif
2249                 return (FC_NOMAP);
2250         }
2251 }
```

*unchanged_portion_omitted_*

```
2274 /*
2275  * duplicate the shared page tables
2276  */
2277 int
2278 segspt_shmdup(struct seg *seg, struct seg *newseg)
2279 {
2280         struct shm_data         *shmd = (struct shm_data *)seg->s_data;
2281         struct anon_map         *amp = shmd->shm_amp;
2282         struct shm_data         *shmd_new;
2283         struct seg              *spt_seg = shmd->shm_sptseg;
2284         struct spt_data         *sptd = spt_seg->s_data;
2285         int                     error = 0;

2287         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
2287         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

2289         shmd_new = kmem_zalloc((sizeof (*shmd_new)), KM_SLEEP);
2290         newseg->s_data = (void *)shmd_new;
2291         shmd_new->shm_sptas = shmd->shm_sptas;
2292         shmd_new->shm_amp = amp;
2293         shmd_new->shm_sptseg = shmd->shm_sptseg;
2294         newseg->s_ops = &segspt_shmops;
2295         newseg->s_szc = seg->s_szc;
2296         ASSERT(seg->s_szc == shmd->shm_sptseg->s_szc);

2298         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2299         amp->refcnt++;
2300         ANON_LOCK_EXIT(&amp->a_rwlock);

2302         if (sptd->spt_flags & SHM_PAGEABLE) {
2303                 shmd_new->shm_vpage = kmem_zalloc(btopr(amp->size), KM_SLEEP);
2304                 shmd_new->shm_lckpgs = 0;
2305                 if (hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
2306                         if ((error = hat_share(newseg->s_as->a_hat,
2307                             newseg->s_base, shmd->shm_sptas->a_hat, SEGSPTADDR,
2308                             seg->s_size, seg->s_szc)) != 0) {
2309                                 kmem_free(shmd_new->shm_vpage,
2310                                     btopr(amp->size));
2311                         }
2312                 }
2313                 return (error);
2314         } else {
```



```
2315                 return (hat_share(newseg->s_as->a_hat, newseg->s_base,
2316                     shmd->shm_sptas->a_hat, SEGSPTADDR, seg->s_size,
2317                     seg->s_szc));

2319         }
2320 }

2322 /*ARGSUSED*/
2323 int
2324 segspt_shmcheckprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
2325 {
2326         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2327         struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;

2329         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2329         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2331         /*
2332          * ISM segment is always rw.
2333          */
2334         return (((sptd->spt_prot & prot) != prot) ? EACCES : 0);
2335 }
```

*unchanged_portion_omitted_*

```
2658 /*ARGSUSED*/
2659 static int
2660 segspt_shmlockop(struct seg *seg, caddr_t addr, size_t len,
2661     int attr, int op, ulong_t *lockmap, size_t pos)
2662 {
2663         struct shm_data *shmd = seg->s_data;
2664         struct seg      *sptseg = shmd->shm_sptseg;
2665         struct spt_data *sptd = sptseg->s_data;
2666         struct kshmid   *sp = sptd->spt_amp->a_sp;
2667         pgcnt_t         npages, a_npages;
2668         page_t          **ppa;
2669         pgcnt_t         an_idx, a_an_idx, ppa_idx;
2670         caddr_t         spt_addr, a_addr;       /* spt and aligned address */
2671         size_t          a_len;                  /* aligned len */
2672         size_t          share_sz;
2673         ulong_t         i;
2674         int             sts = 0;
2675         rctl_qty_t      unlocked = 0;
2676         rctl_qty_t      locked = 0;
2677         struct proc     *p = curproc;
2678         kproject_t      *proj;

2680         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2680         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
2681         ASSERT(sp != NULL);

2683         if ((sptd->spt_flags & SHM_PAGEABLE) == 0) {
2684                 return (0);
2685         }

2687         addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2688         an_idx = seg_page(seg, addr);
2689         npages = btopr(len);

2691         if (an_idx + npages > btopr(shmd->shm_amp->size)) {
2692                 return (ENOMEM);
2693         }

2695         /*
2696          * A shm's project never changes, so no lock needed.
2697          * The shm has a hold on the project, so it will not go away.
2698          * Since we have a mapping to shm within this zone, we know
```

```
2699          * that the zone will not go away.
2700          */
2701         proj = sp->shm_perm.ipc_proj;

2703         if (op == MC_LOCK) {

2705                 /*
2706                  * Need to align addr and size request if they are not
2707                  * aligned so we can always allocate large page(s) however
2708                  * we only lock what was requested in initial request.
2709                  */
2710                 share_sz = page_get_pagesize(sptseg->s_szc);
2711                 a_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), share_sz);
2712                 a_len = P2ROUNDUP((uintptr_t)(((addr + len) - a_addr)),
2713                     share_sz);
2714                 a_npages = btop(a_len);
2715                 a_an_idx = seg_page(seg, a_addr);
2716                 spt_addr = sptseg->s_base + ptob(a_an_idx);
2717                 ppa_idx = an_idx - a_an_idx;

2719                 if ((ppa = kmem_zalloc(((sizeof (page_t *)) * a_npages),
2720                     KM_NOSLEEP)) == NULL) {
2721                         return (ENOMEM);
2722                 }

2724                 /*
2725                  * Don't cache any new pages for IO and
2726                  * flush any cached pages.
2727                  */
2728                 mutex_enter(&sptd->spt_lock);
2729                 if (sptd->spt_ppa != NULL)
2730                         sptd->spt_flags |= DISM_PPA_CHANGED;

2732                 sts = spt_anon_getpages(sptseg, spt_addr, a_len, ppa);
2733                 if (sts != 0) {
2734                         mutex_exit(&sptd->spt_lock);
2735                         kmem_free(ppa, ((sizeof (page_t *)) * a_npages));
2736                         return (sts);
2737                 }

2739                 mutex_enter(&sp->shm_mlock);
2740                 /* enforce locked memory rctl */
2741                 unlocked = spt_unlockedbytes(npages, &ppa[ppa_idx]);

2743                 mutex_enter(&p->p_lock);
2744                 if (rctl_incr_locked_mem(p, proj, unlocked, 0)) {
2745                         mutex_exit(&p->p_lock);
2746                         sts = EAGAIN;
2747                 } else {
2748                         mutex_exit(&p->p_lock);
2749                         sts = spt_lockpages(seg, an_idx, npages,
2750                             &ppa[ppa_idx], lockmap, pos, &locked);

2752                         /*
2753                          * correct locked count if not all pages could be
2754                          * locked
2755                          */
2756                         if ((unlocked - locked) > 0) {
2757                                 rctl_decr_locked_mem(NULL, proj,
2758                                     (unlocked - locked), 0);
2759                         }
2760                 }
2761                 /*
2762                  * unlock pages
2763                  */
2764                 for (i = 0; i < a_npages; i++)
```



```
2765                         page_unlock(ppa[i]);
2766                 if (sptd->spt_ppa != NULL)
2767                         sptd->spt_flags |= DISM_PPA_CHANGED;
2768                 mutex_exit(&sp->shm_mlock);
2769                 mutex_exit(&sptd->spt_lock);

2771                 kmem_free(ppa, ((sizeof (page_t *)) * a_npages));

2773         } else if (op == MC_UNLOCK) { /* unlock */
2774                 page_t          **ppa;

2776                 mutex_enter(&sptd->spt_lock);
2777                 if (shmd->shm_lckpgs == 0) {
2778                         mutex_exit(&sptd->spt_lock);
2779                         return (0);
2780                 }
2781                 /*
2782                  * Don't cache new IO pages.
2783                  */
2784                 if (sptd->spt_ppa != NULL)
2785                         sptd->spt_flags |= DISM_PPA_CHANGED;

2787                 mutex_enter(&sp->shm_mlock);
2788                 sts = spt_unlockpages(seg, an_idx, npages, &unlocked);
2789                 if ((ppa = sptd->spt_ppa) != NULL)
2790                         sptd->spt_flags |= DISM_PPA_CHANGED;
2791                 mutex_exit(&sptd->spt_lock);

2793                 rctl_decr_locked_mem(NULL, proj, unlocked, 0);
2794                 mutex_exit(&sp->shm_mlock);

2796                 if (ppa != NULL)
2797                         seg_ppurge_wiredpp(ppa);
2798         }
2799         return (sts);
2800 }

2802 /*ARGSUSED*/
2803 int
2804 segspt_shmgetprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
2805 {
2806         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2807         struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;
2808         spgcnt_t pgno = seg_page(seg, addr+len) - seg_page(seg, addr) + 1;

2810         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2810         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2812         /*
2813          * ISM segment is always rw.
2814          */
2815         while (--pgno >= 0)
2816                 *protv++ = sptd->spt_prot;
2817         return (0);
2818 }

2820 /*ARGSUSED*/
2821 u_offset_t
2822 segspt_shmgetoffset(struct seg *seg, caddr_t addr)
2823 {
2824         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2824         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2826         /* Offset does not matter in ISM memory */

2828         return ((u_offset_t)0);
```

```
2829 }

2831 /* ARGSUSED */
2832 int
2833 segspt_shmgettype(struct seg *seg, caddr_t addr)
2834 {
2835         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2836         struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;

2838         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2838         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2840         /*
2841          * The shared memory mapping is always MAP_SHARED, SWAP is only
2842          * reserved for DISM
2843          */
2844         return (MAP_SHARED |
2845             ((sptd->spt_flags & SHM_PAGEABLE) ? 0 : MAP_NORESERVE));
2846 }

2848 /*ARGSUSED*/
2849 int
2850 segspt_shmgetvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
2851 {
2852         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2853         struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;

2855         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2855         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2857         *vpp = sptd->spt_vp;
2858         return (0);
2859 }

2861 /*
2862  * We need to wait for pending IO to complete to a DISM segment in order for
2863  * pages to get kicked out of the seg_pcache.  120 seconds should be more
2864  * than enough time to wait.
2865  */
2866 static clock_t spt_pcache_wait = 120;

2868 /*ARGSUSED*/
2869 static int
2870 segspt_shmadvise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
2871 {
2872         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2873         struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;
2874         struct anon_map *amp;
2875         pgcnt_t pg_idx;
2876         ushort_t gen;
2877         clock_t end_lbolt;
2878         int writer;
2879         page_t **ppa;

2881         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2881         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2883         if (behav == MADV_FREE) {
2884                 if ((sptd->spt_flags & SHM_PAGEABLE) == 0)
2885                         return (0);

2887                 amp = sptd->spt_amp;
2888                 pg_idx = seg_page(seg, addr);

2890                 mutex_enter(&sptd->spt_lock);
2891                 if ((ppa = sptd->spt_ppa) == NULL) {
```



```
2892                         mutex_exit(&sptd->spt_lock);
2893                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2894                         anon_disclaim(amp, pg_idx, len);
2895                         ANON_LOCK_EXIT(&amp->a_rwlock);
2896                         return (0);
2897                 }

2899                 sptd->spt_flags |= DISM_PPA_CHANGED;
2900                 gen = sptd->spt_gen;

2902                 mutex_exit(&sptd->spt_lock);

2904                 /*
2905                  * Purge all DISM cached pages
2906                  */
2907                 seg_ppurge_wiredpp(ppa);

2909                 /*
2910                  * Drop the AS_LOCK so that other threads can grab it
2911                  * in the as_pageunlock path and hopefully get the segment
2912                  * kicked out of the seg_pcache.  We bump the shm_softlockcnt
2913                  * to keep this segment resident.
2914                  */
2915                 writer = AS_WRITE_HELD(seg->s_as);
2915                 writer = AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock);
2916                 atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
2917                 AS_LOCK_EXIT(seg->s_as);
2917                 AS_LOCK_EXIT(seg->s_as, &seg->s_as->a_lock);

2919                 mutex_enter(&sptd->spt_lock);

2921                 end_lbolt = ddi_get_lbolt() + (hz * spt_pcache_wait);

2923                 /*
2924                  * Try to wait for pages to get kicked out of the seg_pcache.
2925                  */
2926                 while (sptd->spt_gen == gen &&
2927                     (sptd->spt_flags & DISM_PPA_CHANGED) &&
2928                     ddi_get_lbolt() < end_lbolt) {
2929                         if (!cv_timedwait_sig(&sptd->spt_cv,
2930                             &sptd->spt_lock, end_lbolt)) {
2931                                 break;
2932                         }
2933                 }

2935                 mutex_exit(&sptd->spt_lock);

2937                 /* Regrab the AS_LOCK and release our hold on the segment */
2938                 AS_LOCK_ENTER(seg->s_as, writer ? RW_WRITER : RW_READER);
2938                 AS_LOCK_ENTER(seg->s_as, &seg->s_as->a_lock,
2939                     writer ? RW_WRITER : RW_READER);
2939                 atomic_dec_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
2940                 if (shmd->shm_softlockcnt <= 0) {
2941                         if (AS_ISUNMAPWAIT(seg->s_as)) {
2942                                 mutex_enter(&seg->s_as->a_contents);
2943                                 if (AS_ISUNMAPWAIT(seg->s_as)) {
2944                                         AS_CLRUNMAPWAIT(seg->s_as);
2945                                         cv_broadcast(&seg->s_as->a_cv);
2946                                 }
2947                                 mutex_exit(&seg->s_as->a_contents);
2948                         }
2949                 }

2951                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2952                 anon_disclaim(amp, pg_idx, len);
2953                 ANON_LOCK_EXIT(&amp->a_rwlock);
```

```
2954         } else if (lgrp_optimizations() && (behav == MADV_ACCESS_LWP ||
2955             behav == MADV_ACCESS_MANY || behav == MADV_ACCESS_DEFAULT)) {
2956                 int                     already_set;
2957                 ulong_t                 anon_index;
2958                 lgrp_mem_policy_t       policy;
2959                 caddr_t                 shm_addr;
2960                 size_t                  share_size;
2961                 size_t                  size;
2962                 struct seg              *sptseg = shmd->shm_sptseg;
2963                 caddr_t                 sptseg_addr;

2965                 /*
2966                  * Align address and length to page size of underlying segment
2967                  */
2968                 share_size = page_get_pagesize(shmd->shm_sptseg->s_szc);
2969                 shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), share_size);
2970                 size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)),
2971                     share_size);

2973                 amp = shmd->shm_amp;
2974                 anon_index = seg_page(seg, shm_addr);

2976                 /*
2977                  * And now we may have to adjust size downward if we have
2978                  * exceeded the realsize of the segment or initial anon
2979                  * allocations.
2980                  */
2981                 sptseg_addr = sptseg->s_base + ptob(anon_index);
2982                 if ((sptseg_addr + size) >
2983                     (sptseg->s_base + sptd->spt_realsize))
2984                         size = (sptseg->s_base + sptd->spt_realsize) -
2985                             sptseg_addr;

2987                 /*
2988                  * Set memory allocation policy for this segment
2989                  */
2990                 policy = lgrp_madv_to_policy(behav, len, MAP_SHARED);
2991                 already_set = lgrp_shm_policy_set(policy, amp, anon_index,
2992                     NULL, 0, len);

2994                 /*
2995                  * If random memory allocation policy set already,
2996                  * don't bother reapplying it.
2997                  */
2998                 if (already_set && !LGRP_MEM_POLICY_REAPPLICABLE(policy))
2999                         return (0);

3001                 /*
3002                  * Mark any existing pages in the given range for
3003                  * migration, flushing the I/O page cache, and using
3004                  * underlying segment to calculate anon index and get
3005                  * anonmap and vnode pointer from
3006                  */
3007                 if (shmd->shm_softlockcnt > 0)
3008                         segspt_purge(seg);

3010                 page_mark_migrate(seg, shm_addr, size, amp, 0, NULL, 0, 0);
3011         }

3013         return (0);
3014 }
```

unchanged_portion_omitted_

```
**********************************************************
  284868 Wed Nov 25 13:59:40 2015
new/usr/src/uts/common/vm/seg_vn.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 540 int
 541 segvn_create(struct seg *seg, void *argsp)
 542 {
 543         struct segvn_crargs *a = (struct segvn_crargs *)argsp;
 544         struct segvn_data *svd;
 545         size_t swresv = 0;
 546         struct cred *cred;
 547         struct anon_map *amp;
 548         int error = 0;
 549         size_t pgsz;
 550         lgrp_mem_policy_t mpolicy = LGRP_MEM_POLICY_DEFAULT;
 551         int use_rgn = 0;
 552         int trok = 0;

 554         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
 554         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 556         if (a->type != MAP_PRIVATE && a->type != MAP_SHARED) {
 557                 panic("segvn_create type");
 558                 /*NOTREACHED*/
 559         }

 561         /*
 562          * Check arguments.  If a shared anon structure is given then
 563          * it is illegal to also specify a vp.
 564          */
 565         if (a->amp != NULL && a->vp != NULL) {
 566                 panic("segvn_create anon_map");
 567                 /*NOTREACHED*/
 568         }

 570         if (a->type == MAP_PRIVATE && (a->flags & MAP_TEXT) &&
 571             a->vp != NULL && a->prot == (PROT_USER | PROT_READ | PROT_EXEC) &&
 572             segvn_use_regions) {
 573                 use_rgn = 1;
 574         }

 576         /* MAP_NORESERVE on a MAP_SHARED segment is meaningless. */
 577         if (a->type == MAP_SHARED)
 578                 a->flags &= ~MAP_NORESERVE;

 580         if (a->szc != 0) {
 581                 if (segvn_lpg_disable != 0 || (a->szc == AS_MAP_NO_LPOOB) ||
 582                     (a->amp != NULL && a->type == MAP_PRIVATE) ||
 583                     (a->flags & MAP_NORESERVE) || seg->s_as == &kas) {
 584                         a->szc = 0;
 585                 } else {
 586                         if (a->szc > segvn_maxpgszc)
 587                                 a->szc = segvn_maxpgszc;
 588                         pgsz = page_get_pagesize(a->szc);
 589                         if (!IS_P2ALIGNED(seg->s_base, pgsz) ||
 590                             !IS_P2ALIGNED(seg->s_size, pgsz)) {
 591                                 a->szc = 0;
 592                         } else if (a->vp != NULL) {
 593                                 if (IS_SWAPFSVP(a->vp) || VN_ISKAS(a->vp)) {
 594                                         /*
 595                                          * paranoid check.
 596                                          * hat_page_demote() is not supported
 597                                          * on swapfs pages.
```



```
 598                                          */
 599                                         a->szc = 0;
 600                                 } else if (map_addr_vacalign_check(seg->s_base,
 601                                     a->offset & PAGEMASK)) {
 602                                         a->szc = 0;
 603                                 }
 604                         } else if (a->amp != NULL) {
 605                                 pgcnt_t anum = btopr(a->offset);
 606                                 pgcnt_t pgcnt = page_get_pagecnt(a->szc);
 607                                 if (!IS_P2ALIGNED(anum, pgcnt)) {
 608                                         a->szc = 0;
 609                                 }
 610                         }
 611                 }
 612         }

 614         /*
 615          * If segment may need private pages, reserve them now.
 616          */
 617         if (!(a->flags & MAP_NORESERVE) && ((a->vp == NULL && a->amp == NULL) ||
 618             (a->type == MAP_PRIVATE && (a->prot & PROT_WRITE)))) {
 619                 if (anon_resv_zone(seg->s_size,
 620                     seg->s_as->a_proc->p_zone) == 0)
 621                         return (EAGAIN);
 622                 swresv = seg->s_size;
 623                 TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
 624                     seg, swresv, 1);
 625         }

 627         /*
 628          * Reserve any mapping structures that may be required.
 629          *
 630          * Don't do it for segments that may use regions. It's currently a
 631          * noop in the hat implementations anyway.
 632          */
 633         if (!use_rgn) {
 634                 hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);
 635         }

 637         if (a->cred) {
 638                 cred = a->cred;
 639                 crhold(cred);
 640         } else {
 641                 crhold(cred = CRED());
 642         }

 644         /* Inform the vnode of the new mapping */
 645         if (a->vp != NULL) {
 646                 error = VOP_ADDMAP(a->vp, a->offset & PAGEMASK,
 647                     seg->s_as, seg->s_base, seg->s_size, a->prot,
 648                     a->maxprot, a->type, cred, NULL);
 649                 if (error) {
 650                         if (swresv != 0) {
 651                                 anon_unresv_zone(swresv,
 652                                     seg->s_as->a_proc->p_zone);
 653                                 TRACE_3(TR_FAC_VM, TR_ANON_PROC,
 654                                     "anon proc:%p %lu %u", seg, swresv, 0);
 655                         }
 656                         crfree(cred);
 657                         if (!use_rgn) {
 658                                 hat_unload(seg->s_as->a_hat, seg->s_base,
 659                                     seg->s_size, HAT_UNLOAD_UNMAP);
 660                         }
 661                         return (error);
 662                 }
 663                 /*
```

```
664                  * svntr_hashtab will be NULL if we support shared regions.
665                  */
666                 trok = ((a->flags & MAP_TEXT) &&
667                     (seg->s_size > textrepl_size_thresh ||
668                     (a->flags & _MAP_TEXTREPL)) &&
669                     lgrp_optimizations() && svntr_hashtab != NULL &&
670                     a->type == MAP_PRIVATE && swresv == 0 &&
671                     !(a->flags & MAP_NORESERVE) &&
672                     seg->s_as != &kas && a->vp->v_type == VREG);

674                 ASSERT(!trok || !use_rgn);
675         }

677         /*
678          * MAP_NORESERVE mappings don't count towards the VSZ of a process
679          * until we fault the pages in.
680          */
681         if ((a->vp == NULL || a->vp->v_type != VREG) &&
682             a->flags & MAP_NORESERVE) {
683                 seg->s_as->a_resvsize -= seg->s_size;
684         }

686         /*
687          * If more than one segment in the address space, and they're adjacent
688          * virtually, try to concatenate them.  Don't concatenate if an
689          * explicit anon_map structure was supplied (e.g., SystemV shared
690          * memory) or if we'll use text replication for this segment.
691          */
692         if (a->amp == NULL && !use_rgn && !trok) {
693                 struct seg *pseg, *nseg;
694                 struct segvn_data *psvd, *nsvd;
695                 lgrp_mem_policy_t ppolicy, npolicy;
696                 uint_t  lgrp_mem_policy_flags = 0;
697                 extern lgrp_mem_policy_t lgrp_mem_default_policy;

699                 /*
700                  * Memory policy flags (lgrp_mem_policy_flags) is valid when
701                  * extending stack/heap segments.
702                  */
703                 if ((a->vp == NULL) && (a->type == MAP_PRIVATE) &&
704                     !(a->flags & MAP_NORESERVE) && (seg->s_as != &kas)) {
705                         lgrp_mem_policy_flags = a->lgrp_mem_policy_flags;
706                 } else {
707                         /*
708                          * Get policy when not extending it from another segment
709                          */
710                         mpolicy = lgrp_mem_policy_default(seg->s_size, a->type);
711                 }

713                 /*
714                  * First, try to concatenate the previous and new segments
715                  */
716                 pseg = AS_SEGPREV(seg->s_as, seg);
717                 if (pseg != NULL &&
718                     pseg->s_base + pseg->s_size == seg->s_base &&
719                     pseg->s_ops == &segvn_ops) {
720                         /*
721                          * Get memory allocation policy from previous segment.
722                          * When extension is specified (e.g. for heap) apply
723                          * this policy to the new segment regardless of the
724                          * outcome of segment concatenation.  Extension occurs
725                          * for non-default policy otherwise default policy is
726                          * used and is based on extended segment size.
727                          */
728                         psvd = (struct segvn_data *)pseg->s_data;
729                         ppolicy = psvd->policy_info.mem_policy;
```



```
730                         if (lgrp_mem_policy_flags ==
731                             LGRP_MP_FLAG_EXTEND_UP) {
732                                 if (ppolicy != lgrp_mem_default_policy) {
733                                         mpolicy = ppolicy;
734                                 } else {
735                                         mpolicy = lgrp_mem_policy_default(
736                                             pseg->s_size + seg->s_size,
737                                             a->type);
738                                 }
739                         }

741                         if (mpolicy == ppolicy &&
742                             (pseg->s_size + seg->s_size <=
743                             segvn_comb_thrshld || psvd->amp == NULL) &&
744                             segvn_extend_prev(pseg, seg, a, swresv) == 0) {
745                                 /*
746                                  * success! now try to concatenate
747                                  * with following seg
748                                  */
749                                 crfree(cred);
750                                 nseg = AS_SEGNEXT(pseg->s_as, pseg);
751                                 if (nseg != NULL &&
752                                     nseg != pseg &&
753                                     nseg->s_ops == &segvn_ops &&
754                                     pseg->s_base + pseg->s_size ==
755                                     nseg->s_base)
756                                         (void) segvn_concat(pseg, nseg, 0);
757                                 ASSERT(pseg->s_szc == 0 ||
758                                     (a->szc == pseg->s_szc &&
759                                     IS_P2ALIGNED(pseg->s_base, pgsz) &&
760                                     IS_P2ALIGNED(pseg->s_size, pgsz)));
761                                 return (0);
762                         }
763                 }

765                 /*
766                  * Failed, so try to concatenate with following seg
767                  */
768                 nseg = AS_SEGNEXT(seg->s_as, seg);
769                 if (nseg != NULL &&
770                     seg->s_base + seg->s_size == nseg->s_base &&
771                     nseg->s_ops == &segvn_ops) {
772                         /*
773                          * Get memory allocation policy from next segment.
774                          * When extension is specified (e.g. for stack) apply
775                          * this policy to the new segment regardless of the
776                          * outcome of segment concatenation.  Extension occurs
777                          * for non-default policy otherwise default policy is
778                          * used and is based on extended segment size.
779                          */
780                         nsvd = (struct segvn_data *)nseg->s_data;
781                         npolicy = nsvd->policy_info.mem_policy;
782                         if (lgrp_mem_policy_flags ==
783                             LGRP_MP_FLAG_EXTEND_DOWN) {
784                                 if (npolicy != lgrp_mem_default_policy) {
785                                         mpolicy = npolicy;
786                                 } else {
787                                         mpolicy = lgrp_mem_policy_default(
788                                             nseg->s_size + seg->s_size,
789                                             a->type);
790                                 }
791                         }

793                         if (mpolicy == npolicy &&
794                             segvn_extend_next(seg, nseg, a, swresv) == 0) {
795                                 crfree(cred);
```

```
796                                 ASSERT(nseg->s_szc == 0 ||
797                                     (a->szc == nseg->s_szc &&
798                                     IS_P2ALIGNED(nseg->s_base, pgsz) &&
799                                     IS_P2ALIGNED(nseg->s_size, pgsz)));
800                                 return (0);
801                         }
802                 }
803         }

805         if (a->vp != NULL) {
806                 VN_HOLD(a->vp);
807                 if (a->type == MAP_SHARED)
808                         lgrp_shm_policy_init(NULL, a->vp);
809         }
810         svd = kmem_cache_alloc(segvn_cache, KM_SLEEP);

812         seg->s_ops = &segvn_ops;
813         seg->s_data = (void *)svd;
814         seg->s_szc = a->szc;

816         svd->seg = seg;
817         svd->vp = a->vp;
818         /*
819          * Anonymous mappings have no backing file so the offset is meaningless.
820          */
821         svd->offset = a->vp ? (a->offset & PAGEMASK) : 0;
822         svd->prot = a->prot;
823         svd->maxprot = a->maxprot;
824         svd->pageprot = 0;
825         svd->type = a->type;
826         svd->vpage = NULL;
827         svd->cred = cred;
828         svd->advice = MADV_NORMAL;
829         svd->pageadvice = 0;
830         svd->flags = (ushort_t)a->flags;
831         svd->softlockcnt = 0;
832         svd->softlockcnt_sbase = 0;
833         svd->softlockcnt_send = 0;
834         svd->svn_inz = 0;
835         svd->rcookie = HAT_INVALID_REGION_COOKIE;
836         svd->pageswap = 0;

838         if (a->szc != 0 && a->vp != NULL) {
839                 segvn_setvnode_mpss(a->vp);
840         }
841         if (svd->type == MAP_SHARED && svd->vp != NULL &&
842             (svd->vp->v_flag & VVMEXEC) && (svd->prot & PROT_WRITE)) {
843                 ASSERT(vn_is_mapped(svd->vp, V_WRITE));
844                 segvn_inval_trcache(svd->vp);
845         }

847         amp = a->amp;
848         if ((svd->amp = amp) == NULL) {
849                 svd->anon_index = 0;
850                 if (svd->type == MAP_SHARED) {
851                         svd->swresv = 0;
852                         /*
853                          * Shared mappings to a vp need no other setup.
854                          * If we have a shared mapping to an anon_map object
855                          * which hasn't been allocated yet,  allocate the
856                          * struct now so that it will be properly shared
857                          * by remembering the swap reservation there.
858                          */
859                         if (a->vp == NULL) {
860                                 svd->amp = anonmap_alloc(seg->s_size, swresv,
861                                     ANON_SLEEP);
```



```
862                                 svd->amp->a_szc = seg->s_szc;
863                         }
864                 } else {
865                         /*
866                          * Private mapping (with or without a vp).
867                          * Allocate anon_map when needed.
868                          */
869                         svd->swresv = swresv;
870                 }
871         } else {
872                 pgcnt_t anon_num;

874                 /*
875                  * Mapping to an existing anon_map structure without a vp.
876                  * For now we will insure that the segment size isn't larger
877                  * than the size - offset gives us.  Later on we may wish to
878                  * have the anon array dynamically allocated itself so that
879                  * we don't always have to allocate all the anon pointer slots.
880                  * This of course involves adding extra code to check that we
881                  * aren't trying to use an anon pointer slot beyond the end
882                  * of the currently allocated anon array.
883                  */
884                 if ((amp->size - a->offset) < seg->s_size) {
885                         panic("segvn_create anon_map size");
886                         /*NOTREACHED*/
887                 }

889                 anon_num = btopr(a->offset);

891                 if (a->type == MAP_SHARED) {
892                         /*
893                          * SHARED mapping to a given anon_map.
894                          */
895                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
896                         amp->refcnt++;
897                         if (a->szc > amp->a_szc) {
898                                 amp->a_szc = a->szc;
899                         }
900                         ANON_LOCK_EXIT(&amp->a_rwlock);
901                         svd->anon_index = anon_num;
902                         svd->swresv = 0;
903                 } else {
904                         /*
905                          * PRIVATE mapping to a given anon_map.
906                          * Make sure that all the needed anon
907                          * structures are created (so that we will
908                          * share the underlying pages if nothing
909                          * is written by this mapping) and then
910                          * duplicate the anon array as is done
911                          * when a privately mapped segment is dup'ed.
912                          */
913                         struct anon *ap;
914                         caddr_t addr;
915                         caddr_t eaddr;
916                         ulong_t anon_idx;
917                         int hat_flag = HAT_LOAD;

919                         if (svd->flags & MAP_TEXT) {
920                                 hat_flag |= HAT_LOAD_TEXT;
921                         }

923                         svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
924                         svd->amp->a_szc = seg->s_szc;
925                         svd->anon_index = 0;
926                         svd->swresv = swresv;
```

```
928                         /*
929                          * Prevent 2 threads from allocating anon
930                          * slots simultaneously.
931                          */
932                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
933                         eaddr = seg->s_base + seg->s_size;
934
935                         for (anon_idx = anon_num, addr = seg->s_base;
936                             addr < eaddr; addr += PAGESIZE, anon_idx++) {
937                                 page_t *pp;
938
939                                 if ((ap = anon_get_ptr(amp->ahp,
940                                     anon_idx)) != NULL)
941                                         continue;
942
943                                 /*
944                                  * Allocate the anon struct now.
945                                  * Might as well load up translation
946                                  * to the page while we're at it...
947                                  */
948                                 pp = anon_zero(seg, addr, &ap, cred);
949                                 if (ap == NULL || pp == NULL) {
950                                         panic("segvn_create anon_zero");
951                                         /*NOTREACHED*/
952                                 }
953
954                                 /*
955                                  * Re-acquire the anon_map lock and
956                                  * initialize the anon array entry.
957                                  */
958                                 ASSERT(anon_get_ptr(amp->ahp,
959                                     anon_idx) == NULL);
960                                 (void) anon_set_ptr(amp->ahp, anon_idx, ap,
961                                     ANON_SLEEP);
962
963                                 ASSERT(seg->s_szc == 0);
964                                 ASSERT(!IS_VMODSORT(pp->p_vnode));
965
966                                 ASSERT(use_rgn == 0);
967                                 hat_memload(seg->s_as->a_hat, addr, pp,
968                                     svd->prot & ~PROT_WRITE, hat_flag);
969
970                                 page_unlock(pp);
971                         }
972                         ASSERT(seg->s_szc == 0);
973                         anon_dup(amp->ahp, anon_num, svd->amp->ahp,
974                             0, seg->s_size);
975                         ANON_LOCK_EXIT(&amp->a_rwlock);
976                 }
977         }
978
979         /*
980          * Set default memory allocation policy for segment
981          *
982          * Always set policy for private memory at least for initialization
983          * even if this is a shared memory segment
984          */
985         (void) lgrp_privm_policy_set(mpolicy, &svd->policy_info, seg->s_size);
986
987         if (svd->type == MAP_SHARED)
988                 (void) lgrp_shm_policy_set(mpolicy, svd->amp, svd->anon_index,
989                     svd->vp, svd->offset, seg->s_size);
990
991         if (use_rgn) {
992                 ASSERT(!trok);
993                 ASSERT(svd->amp == NULL);
```



```
994                 svd->rcookie = hat_join_region(seg->s_as->a_hat, seg->s_base,
995                     seg->s_size, (void *)svd->vp, svd->offset, svd->prot,
996                     (uchar_t)seg->s_szc, segvn_hat_rgn_unload_callback,
997                     HAT_REGION_TEXT);
998         }
999
1000        ASSERT(!trok || !(svd->prot & PROT_WRITE));
1001        svd->tr_state = trok ? SEGVN_TR_INIT : SEGVN_TR_OFF;
1002
1003        return (0);
1004 }
1005
1006 /*
1007  * Concatenate two existing segments, if possible.
1008  * Return 0 on success, -1 if two segments are not compatible
1009  * or -2 on memory allocation failure.
1010  * If amp_cat == 1 then try and concat segments with anon maps
1011  */
1012 static int
1013 segvn_concat(struct seg *seg1, struct seg *seg2, int amp_cat)
1014 {
1015         struct segvn_data *svd1 = seg1->s_data;
1016         struct segvn_data *svd2 = seg2->s_data;
1017         struct anon_map *amp1 = svd1->amp;
1018         struct anon_map *amp2 = svd2->amp;
1019         struct vpage *vpage1 = svd1->vpage;
1020         struct vpage *vpage2 = svd2->vpage, *nvpage = NULL;
1021         size_t size, nvpsize;
1022         pgcnt_t npages1, npages2;
1023
1024         ASSERT(seg1->s_as && seg2->s_as && seg1->s_as == seg2->s_as);
1025         ASSERT(AS_WRITE_HELD(seg1->s_as));
1025         ASSERT(AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));
1026         ASSERT(seg1->s_ops == seg2->s_ops);
1027
1028         if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie) ||
1029             HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1030                 return (-1);
1031         }
1032
1033         /* both segments exist, try to merge them */
1034 #define incompat(x)     (svd1->x != svd2->x)
1035         if (incompat(vp) || incompat(maxprot) ||
1036             (!svd1->pageadvice && !svd2->pageadvice && incompat(advice)) ||
1037             (!svd1->pageprot && !svd2->pageprot && incompat(prot)) ||
1038             incompat(type) || incompat(cred) || incompat(flags) ||
1039             seg1->s_szc != seg2->s_szc || incompat(policy_info.mem_policy) ||
1040             (svd2->softlockcnt > 0) || svd1->softlockcnt_send > 0)
1041                 return (-1);
1042 #undef incompat
1043
1044         /*
1045          * vp == NULL implies zfod, offset doesn't matter
1046          */
1047         if (svd1->vp != NULL &&
1048             svd1->offset + seg1->s_size != svd2->offset) {
1049                 return (-1);
1050         }
1051
1052         /*
1053          * Don't concatenate if either segment uses text replication.
1054          */
1055         if (svd1->tr_state != SEGVN_TR_OFF || svd2->tr_state != SEGVN_TR_OFF) {
1056                 return (-1);
1057         }
```

```
1059         /*
1060          * Fail early if we're not supposed to concatenate
1061          * segments with non NULL amp.
1062          */
1063         if (amp_cat == 0 && (amp1 != NULL || amp2 != NULL)) {
1064                 return (-1);
1065         }

1067         if (svd1->vp == NULL && svd1->type == MAP_SHARED) {
1068                 if (amp1 != amp2) {
1069                         return (-1);
1070                 }
1071                 if (amp1 != NULL && svd1->anon_index + btop(seg1->s_size) !=
1072                     svd2->anon_index) {
1073                         return (-1);
1074                 }
1075                 ASSERT(amp1 == NULL || amp1->refcnt >= 2);
1076         }

1078         /*
1079          * If either seg has vpages, create a new merged vpage array.
1080          */
1081         if (vpage1 != NULL || vpage2 != NULL) {
1082                 struct vpage *vp, *evp;

1084                 npages1 = seg_pages(seg1);
1085                 npages2 = seg_pages(seg2);
1086                 nvpsize = vpgtob(npages1 + npages2);

1088                 if ((nvpage = kmem_zalloc(nvpsize, KM_NOSLEEP)) == NULL) {
1089                         return (-2);
1090                 }

1092                 if (vpage1 != NULL) {
1093                         bcopy(vpage1, nvpage, vpgtob(npages1));
1094                 } else {
1095                         evp = nvpage + npages1;
1096                         for (vp = nvpage; vp < evp; vp++) {
1097                                 VPP_SETPROT(vp, svd1->prot);
1098                                 VPP_SETADVICE(vp, svd1->advice);
1099                         }
1100                 }

1102                 if (vpage2 != NULL) {
1103                         bcopy(vpage2, nvpage + npages1, vpgtob(npages2));
1104                 } else {
1105                         evp = nvpage + npages1 + npages2;
1106                         for (vp = nvpage + npages1; vp < evp; vp++) {
1107                                 VPP_SETPROT(vp, svd2->prot);
1108                                 VPP_SETADVICE(vp, svd2->advice);
1109                         }
1110                 }

1112                 if (svd2->pageswap && (!svd1->pageswap && svd1->swresv)) {
1113                         ASSERT(svd1->swresv == seg1->s_size);
1114                         ASSERT(!(svd1->flags & MAP_NORESERVE));
1115                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1116                         evp = nvpage + npages1;
1117                         for (vp = nvpage; vp < evp; vp++) {
1118                                 VPP_SETSWAPRES(vp);
1119                         }
1120                 }

1122                 if (svd1->pageswap && (!svd2->pageswap && svd2->swresv)) {
1123                         ASSERT(svd2->swresv == seg2->s_size);
1124                         ASSERT(!(svd1->flags & MAP_NORESERVE));
```



```
1125                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1126                         vp = nvpage + npages1;
1127                         evp = vp + npages2;
1128                         for (; vp < evp; vp++) {
1129                                 VPP_SETSWAPRES(vp);
1130                         }
1131                 }
1132         }
1133         ASSERT((vpage1 != NULL || vpage2 != NULL) ||
1134             (svd1->pageswap == 0 && svd2->pageswap == 0));

1136         /*
1137          * If either segment has private pages, create a new merged anon
1138          * array. If mergeing shared anon segments just decrement anon map's
1139          * refcnt.
1140          */
1141         if (amp1 != NULL && svd1->type == MAP_SHARED) {
1142                 ASSERT(amp1 == amp2 && svd1->vp == NULL);
1143                 ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1144                 ASSERT(amp1->refcnt >= 2);
1145                 amp1->refcnt--;
1146                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1147                 svd2->amp = NULL;
1148         } else if (amp1 != NULL || amp2 != NULL) {
1149                 struct anon_hdr *nahp;
1150                 struct anon_map *namp = NULL;
1151                 size_t asize;

1153                 ASSERT(svd1->type == MAP_PRIVATE);

1155                 asize = seg1->s_size + seg2->s_size;
1156                 if ((nahp = anon_create(btop(asize), ANON_NOSLEEP)) == NULL) {
1157                         if (nvpage != NULL) {
1158                                 kmem_free(nvpage, nvpsize);
1159                         }
1160                         return (-2);
1161                 }
1162                 if (amp1 != NULL) {
1163                         /*
1164                          * XXX anon rwlock is not really needed because
1165                          * this is a private segment and we are writers.
1166                          */
1167                         ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1168                         ASSERT(amp1->refcnt == 1);
1169                         if (anon_copy_ptr(amp1->ahp, svd1->anon_index,
1170                             nahp, 0, btop(seg1->s_size), ANON_NOSLEEP)) {
1171                                 anon_release(nahp, btop(asize));
1172                                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1173                                 if (nvpage != NULL) {
1174                                         kmem_free(nvpage, nvpsize);
1175                                 }
1176                                 return (-2);
1177                         }
1178                 }
1179                 if (amp2 != NULL) {
1180                         ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
1181                         ASSERT(amp2->refcnt == 1);
1182                         if (anon_copy_ptr(amp2->ahp, svd2->anon_index,
1183                             nahp, btop(seg1->s_size), btop(seg2->s_size),
1184                             ANON_NOSLEEP)) {
1185                                 anon_release(nahp, btop(asize));
1186                                 ANON_LOCK_EXIT(&amp2->a_rwlock);
1187                                 if (amp1 != NULL) {
1188                                         ANON_LOCK_EXIT(&amp1->a_rwlock);
1189                                 }
1190                                 if (nvpage != NULL) {
```

```
1191                                    kmem_free(nvpage, nvpsize);
1192                            }
1193                            return (-2);
1194                    }
1195            }
1196            if (amp1 != NULL) {
1197                    namp = amp1;
1198                    anon_release(amp1->ahp, btop(amp1->size));
1199            }
1200            if (amp2 != NULL) {
1201                    if (namp == NULL) {
1202                            ASSERT(amp1 == NULL);
1203                            namp = amp2;
1204                            anon_release(amp2->ahp, btop(amp2->size));
1205                    } else {
1206                            amp2->refcnt--;
1207                            ANON_LOCK_EXIT(&amp2->a_rwlock);
1208                            anonmap_free(amp2);
1209                    }
1210                    svd2->amp = NULL; /* needed for seg_free */
1211            }
1212            namp->ahp = nahp;
1213            namp->size = asize;
1214            svd1->amp = namp;
1215            svd1->anon_index = 0;
1216            ANON_LOCK_EXIT(&namp->a_rwlock);
1217        }
1218        /*
1219         * Now free the old vpage structures.
1220         */
1221        if (nvpage != NULL) {
1222                if (vpage1 != NULL) {
1223                        kmem_free(vpage1, vpgtob(npages1));
1224                }
1225                if (vpage2 != NULL) {
1226                        svd2->vpage = NULL;
1227                        kmem_free(vpage2, vpgtob(npages2));
1228                }
1229                if (svd2->pageprot) {
1230                        svd1->pageprot = 1;
1231                }
1232                if (svd2->pageadvice) {
1233                        svd1->pageadvice = 1;
1234                }
1235                if (svd2->pageswap) {
1236                        svd1->pageswap = 1;
1237                }
1238                svd1->vpage = nvpage;
1239        }

1241        /* all looks ok, merge segments */
1242        svd1->swresv += svd2->swresv;
1243        svd2->swresv = 0;  /* so seg_free doesn't release swap space */
1244        size = seg2->s_size;
1245        seg_free(seg2);
1246        seg1->s_size += size;
1247        return (0);
1248 }

1250 /*
1251  * Extend the previous segment (seg1) to include the
1252  * new segment (seg2 + a), if possible.
1253  * Return 0 on success.
1254  */
1255 static int
1256 segvn_extend_prev(seg1, seg2, a, swresv)
```



```
1257         struct seg *seg1, *seg2;
1258         struct segvn_crargs *a;
1259         size_t swresv;
1260 {
1261         struct segvn_data *svd1 = (struct segvn_data *)seg1->s_data;
1262         size_t size;
1263         struct anon_map *amp1;
1264         struct vpage *new_vpage;

1266         /*
1267          * We don't need any segment level locks for "segvn" data
1268          * since the address space is "write" locked.
1269          */
1270         ASSERT(seg1->s_as && AS_WRITE_HELD(seg1->s_as));
1270         ASSERT(seg1->s_as && AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));

1272         if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie)) {
1273                 return (-1);
1274         }

1276         /* second segment is new, try to extend first */
1277         /* XXX - should also check cred */
1278         if (svd1->vp != a->vp || svd1->maxprot != a->maxprot ||
1279             (!svd1->pageprot && (svd1->prot != a->prot)) ||
1280             svd1->type != a->type || svd1->flags != a->flags ||
1281             seg1->s_szc != a->szc || svd1->softlockcnt_send > 0)
1282                 return (-1);

1284         /* vp == NULL implies zfod, offset doesn't matter */
1285         if (svd1->vp != NULL &&
1286             svd1->offset + seg1->s_size != (a->offset & PAGEMASK))
1287                 return (-1);

1289         if (svd1->tr_state != SEGVN_TR_OFF) {
1290                 return (-1);
1291         }

1293         amp1 = svd1->amp;
1294         if (amp1) {
1295                 pgcnt_t newpgs;

1297                 /*
1298                  * Segment has private pages, can data structures
1299                  * be expanded?
1300                  *
1301                  * Acquire the anon_map lock to prevent it from changing,
1302                  * if it is shared.  This ensures that the anon_map
1303                  * will not change while a thread which has a read/write
1304                  * lock on an address space references it.
1305                  * XXX - Don't need the anon_map lock at all if "refcnt"
1306                  * is 1.
1307                  *
1308                  * Can't grow a MAP_SHARED segment with an anonmap because
1309                  * there may be existing anon slots where we want to extend
1310                  * the segment and we wouldn't know what to do with them
1311                  * (e.g., for tmpfs right thing is to just leave them there,
1312                  * for /dev/zero they should be cleared out).
1313                  */
1314                 if (svd1->type == MAP_SHARED)
1315                         return (-1);

1317                 ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1318                 if (amp1->refcnt > 1) {
1319                         ANON_LOCK_EXIT(&amp1->a_rwlock);
1320                         return (-1);
1321                 }
```

```
1322                 newpgs = anon_grow(amp1->ahp, &svd1->anon_index,
1323                     btop(seg1->s_size), btop(seg2->s_size), ANON_NOSLEEP);

1325                 if (newpgs == 0) {
1326                         ANON_LOCK_EXIT(&amp1->a_rwlock);
1327                         return (-1);
1328                 }
1329                 amp1->size = ptob(newpgs);
1330                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1331         }
1332         if (svd1->vpage != NULL) {
1333                 struct vpage *vp, *evp;
1334                 new_vpage =
1335                     kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
1336                         KM_NOSLEEP);
1337                 if (new_vpage == NULL)
1338                         return (-1);
1339                 bcopy(svd1->vpage, new_vpage, vpgtob(seg_pages(seg1)));
1340                 kmem_free(svd1->vpage, vpgtob(seg_pages(seg1)));
1341                 svd1->vpage = new_vpage;

1343                 vp = new_vpage + seg_pages(seg1);
1344                 evp = vp + seg_pages(seg2);
1345                 for (; vp < evp; vp++)
1346                         VPP_SETPROT(vp, a->prot);
1347                 if (svd1->pageswap && swresv) {
1348                         ASSERT(!(svd1->flags & MAP_NORESERVE));
1349                         ASSERT(swresv == seg2->s_size);
1350                         vp = new_vpage + seg_pages(seg1);
1351                         for (; vp < evp; vp++) {
1352                                 VPP_SETSWAPRES(vp);
1353                         }
1354                 }
1355         }
1356         ASSERT(svd1->vpage != NULL || svd1->pageswap == 0);
1357         size = seg2->s_size;
1358         seg_free(seg2);
1359         seg1->s_size += size;
1360         svd1->swresv += swresv;
1361         if (svd1->pageprot && (a->prot & PROT_WRITE) &&
1362             svd1->type == MAP_SHARED && svd1->vp != NULL &&
1363             (svd1->vp->v_flag & VVMEXEC)) {
1364                 ASSERT(vn_is_mapped(svd1->vp, V_WRITE));
1365                 segvn_inval_trcache(svd1->vp);
1366         }
1367         return (0);
1368 }

1370 /*
1371  * Extend the next segment (seg2) to include the
1372  * new segment (seg1 + a), if possible.
1373  * Return 0 on success.
1374  */
1375 static int
1376 segvn_extend_next(
1377         struct seg *seg1,
1378         struct seg *seg2,
1379         struct segvn_crargs *a,
1380         size_t swresv)
1381 {
1382         struct segvn_data *svd2 = (struct segvn_data *)seg2->s_data;
1383         size_t size;
1384         struct anon_map *amp2;
1385         struct vpage *new_vpage;

1387         /*
```



```
1388          * We don't need any segment level locks for "segvn" data
1389          * since the address space is "write" locked.
1390          */
1391         ASSERT(seg2->s_as && AS_WRITE_HELD(seg2->s_as));
1391         ASSERT(seg2->s_as && AS_WRITE_HELD(seg2->s_as, &seg2->s_as->a_lock));

1393         if (HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1394                 return (-1);
1395         }

1397         /* first segment is new, try to extend second */
1398         /* XXX - should also check cred */
1399         if (svd2->vp != a->vp || svd2->maxprot != a->maxprot ||
1400             (!svd2->pageprot && (svd2->prot != a->prot)) ||
1401             svd2->type != a->type || svd2->flags != a->flags ||
1402             seg2->s_szc != a->szc || svd2->softlockcnt_sbase > 0)
1403                 return (-1);
1404         /* vp == NULL implies zfod, offset doesn't matter */
1405         if (svd2->vp != NULL &&
1406             (a->offset & PAGEMASK) + seg1->s_size != svd2->offset)
1407                 return (-1);

1409         if (svd2->tr_state != SEGVN_TR_OFF) {
1410                 return (-1);
1411         }

1413         amp2 = svd2->amp;
1414         if (amp2) {
1415                 pgcnt_t newpgs;

1417                 /*
1418                  * Segment has private pages, can data structures
1419                  * be expanded?
1420                  *
1421                  * Acquire the anon_map lock to prevent it from changing,
1422                  * if it is shared.  This ensures that the anon_map
1423                  * will not change while a thread which has a read/write
1424                  * lock on an address space references it.
1425                  *
1426                  * XXX - Don't need the anon_map lock at all if "refcnt"
1427                  * is 1.
1428                  */
1429                 if (svd2->type == MAP_SHARED)
1430                         return (-1);

1432                 ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
1433                 if (amp2->refcnt > 1) {
1434                         ANON_LOCK_EXIT(&amp2->a_rwlock);
1435                         return (-1);
1436                 }
1437                 newpgs = anon_grow(amp2->ahp, &svd2->anon_index,
1438                     btop(seg2->s_size), btop(seg1->s_size),
1439                     ANON_NOSLEEP | ANON_GROWDOWN);

1441                 if (newpgs == 0) {
1442                         ANON_LOCK_EXIT(&amp2->a_rwlock);
1443                         return (-1);
1444                 }
1445                 amp2->size = ptob(newpgs);
1446                 ANON_LOCK_EXIT(&amp2->a_rwlock);
1447         }
1448         if (svd2->vpage != NULL) {
1449                 struct vpage *vp, *evp;
1450                 new_vpage =
1451                     kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
1452                     KM_NOSLEEP);
```

```
1453                 if (new_vpage == NULL) {
1454                         /* Not merging segments so adjust anon_index back */
1455                         if (amp2)
1456                                 svd2->anon_index += seg_pages(seg1);
1457                         return (-1);
1458                 }
1459                 bcopy(svd2->vpage, new_vpage + seg_pages(seg1),
1460                     vpgtob(seg_pages(seg2)));
1461                 kmem_free(svd2->vpage, vpgtob(seg_pages(seg2)));
1462                 svd2->vpage = new_vpage;

1464                 vp = new_vpage;
1465                 evp = vp + seg_pages(seg1);
1466                 for (; vp < evp; vp++)
1467                         VPP_SETPROT(vp, a->prot);
1468                 if (svd2->pageswap && swresv) {
1469                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1470                         ASSERT(swresv == seg1->s_size);
1471                         vp = new_vpage;
1472                         for (; vp < evp; vp++) {
1473                                 VPP_SETSWAPRES(vp);
1474                         }
1475                 }
1476         }
1477         ASSERT(svd2->vpage != NULL || svd2->pageswap == 0);
1478         size = seg1->s_size;
1479         seg_free(seg1);
1480         seg2->s_size += size;
1481         seg2->s_base -= size;
1482         svd2->offset -= size;
1483         svd2->swresv += swresv;
1484         if (svd2->pageprot && (a->prot & PROT_WRITE) &&
1485             svd2->type == MAP_SHARED && svd2->vp != NULL &&
1486             (svd2->vp->v_flag & VVMEXEC)) {
1487                 ASSERT(vn_is_mapped(svd2->vp, V_WRITE));
1488                 segvn_inval_trcache(svd2->vp);
1489         }
1490         return (0);
1491 }
```

*_____unchanged_portion_omitted_*

```
1568 static int
1569 segvn_dup(struct seg *seg, struct seg *newseg)
1570 {
1571         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1572         struct segvn_data *newsvd;
1573         pgcnt_t npages = seg_pages(seg);
1574         int error = 0;
1575         size_t len;
1576         struct anon_map *amp;

1578         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1578         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
1579         ASSERT(newseg->s_as->a_proc->p_parent == curproc);

1581         /*
1582          * If segment has anon reserved, reserve more for the new seg.
1583          * For a MAP_NORESERVE segment swresv will be a count of all the
1584          * allocated anon slots; thus we reserve for the child as many slots
1585          * as the parent has allocated. This semantic prevents the child or
1586          * parent from dieing during a copy-on-write fault caused by trying
1587          * to write a shared pre-existing anon page.
1588          */
1589         if ((len = svd->swresv) != 0) {
1590                 if (anon_resv(svd->swresv) == 0)
1591                         return (ENOMEM);
```



```
1593                 TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
1594                     seg, len, 0);
1595         }

1597         newsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);

1599         newseg->s_ops = &segvn_ops;
1600         newseg->s_data = (void *)newsvd;
1601         newseg->s_szc = seg->s_szc;

1603         newsvd->seg = newseg;
1604         if ((newsvd->vp = svd->vp) != NULL) {
1605                 VN_HOLD(svd->vp);
1606                 if (svd->type == MAP_SHARED)
1607                         lgrp_shm_policy_init(NULL, svd->vp);
1608         }
1609         newsvd->offset = svd->offset;
1610         newsvd->prot = svd->prot;
1611         newsvd->maxprot = svd->maxprot;
1612         newsvd->pageprot = svd->pageprot;
1613         newsvd->type = svd->type;
1614         newsvd->cred = svd->cred;
1615         crhold(newsvd->cred);
1616         newsvd->advice = svd->advice;
1617         newsvd->pageadvice = svd->pageadvice;
1618         newsvd->svn_inz = svd->svn_inz;
1619         newsvd->swresv = svd->swresv;
1620         newsvd->pageswap = svd->pageswap;
1621         newsvd->flags = svd->flags;
1622         newsvd->softlockcnt = 0;
1623         newsvd->softlockcnt_sbase = 0;
1624         newsvd->softlockcnt_send = 0;
1625         newsvd->policy_info = svd->policy_info;
1626         newsvd->rcookie = HAT_INVALID_REGION_COOKIE;

1628         if ((amp = svd->amp) == NULL || svd->tr_state == SEGVN_TR_ON) {
1629                 /*
1630                  * Not attaching to a shared anon object.
1631                  */
1632                 ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie) ||
1633                     svd->tr_state == SEGVN_TR_OFF);
1634                 if (svd->tr_state == SEGVN_TR_ON) {
1635                         ASSERT(newsvd->vp != NULL && amp != NULL);
1636                         newsvd->tr_state = SEGVN_TR_INIT;
1637                 } else {
1638                         newsvd->tr_state = svd->tr_state;
1639                 }
1640                 newsvd->amp = NULL;
1641                 newsvd->anon_index = 0;
1642         } else {
1643                 /* regions for now are only used on pure vnode segments */
1644                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
1645                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1646                 newsvd->tr_state = SEGVN_TR_OFF;
1647                 if (svd->type == MAP_SHARED) {
1648                         ASSERT(svd->svn_inz == SEGVN_INZ_NONE);
1649                         newsvd->amp = amp;
1650                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
1651                         amp->refcnt++;
1652                         ANON_LOCK_EXIT(&amp->a_rwlock);
1653                         newsvd->anon_index = svd->anon_index;
1654                 } else {
1655                         int reclaim = 1;

1657                         /*
```

```
1658                          * Allocate and initialize new anon_map structure.
1659                          */
1660                         newsvd->amp = anonmap_alloc(newseg->s_size, 0,
1661                             ANON_SLEEP);
1662                         newsvd->amp->a_szc = newseg->s_szc;
1663                         newsvd->anon_index = 0;
1664                         ASSERT(svd->svn_inz == SEGVN_INZ_NONE ||
1665                             svd->svn_inz == SEGVN_INZ_ALL ||
1666                             svd->svn_inz == SEGVN_INZ_VPP);

1668                         /*
1669                          * We don't have to acquire the anon_map lock
1670                          * for the new segment (since it belongs to an
1671                          * address space that is still not associated
1672                          * with any process), or the segment in the old
1673                          * address space (since all threads in it
1674                          * are stopped while duplicating the address space).
1675                          */

1677                         /*
1678                          * The goal of the following code is to make sure that
1679                          * softlocked pages do not end up as copy on write
1680                          * pages.  This would cause problems where one
1681                          * thread writes to a page that is COW and a different
1682                          * thread in the same process has softlocked it.  The
1683                          * softlock lock would move away from this process
1684                          * because the write would cause this process to get
1685                          * a copy (without the softlock).
1686                          *
1687                          * The strategy here is to just break the
1688                          * sharing on pages that could possibly be
1689                          * softlocked.
1690                          *
1691                          * In addition, if any pages have been marked that they
1692                          * should be inherited as zero, then we immediately go
1693                          * ahead and break COW and zero them. In the case of a
1694                          * softlocked page that should be inherited zero, we
1695                          * break COW and just get a zero page.
1696                          */
1697 retry:
1698                         if (svd->softlockcnt ||
1699                             svd->svn_inz != SEGVN_INZ_NONE) {
1700                                 /*
1701                                  * The softlock count might be non zero
1702                                  * because some pages are still stuck in the
1703                                  * cache for lazy reclaim. Flush the cache
1704                                  * now. This should drop the count to zero.
1705                                  * [or there is really I/O going on to these
1706                                  * pages]. Note, we have the writers lock so
1707                                  * nothing gets inserted during the flush.
1708                                  */
1709                                 if (svd->softlockcnt && reclaim == 1) {
1710                                         segvn_purge(seg);
1711                                         reclaim = 0;
1712                                         goto retry;
1713                                 }

1715                                 error = segvn_dup_pages(seg, newseg);
1716                                 if (error != 0) {
1717                                         newsvd->vpage = NULL;
1718                                         goto out;
1719                                 }
1720                         } else {        /* common case */
1721                                 if (seg->s_szc != 0) {
1722                                         /*
1723                                          * If at least one of anon slots of a
```



```
1724                                          * large page exists then make sure
1725                                          * all anon slots of a large page
1726                                          * exist to avoid partial cow sharing
1727                                          * of a large page in the future.
1728                                          */
1729                                         anon_dup_fill_holes(amp->ahp,
1730                                             svd->anon_index, newsvd->amp->ahp,
1731                                             0, seg->s_size, seg->s_szc,
1732                                             svd->vp != NULL);
1733                                 } else {
1734                                         anon_dup(amp->ahp, svd->anon_index,
1735                                             newsvd->amp->ahp, 0, seg->s_size);
1736                                 }

1738                                 hat_clrattr(seg->s_as->a_hat, seg->s_base,
1739                                     seg->s_size, PROT_WRITE);
1740                         }
1741                 }
1742         }
1743         /*
1744          * If necessary, create a vpage structure for the new segment.
1745          * Do not copy any page lock indications.
1746          */
1747         if (svd->vpage != NULL) {
1748                 uint_t i;
1749                 struct vpage *ovp = svd->vpage;
1750                 struct vpage *nvp;

1752                 nvp = newsvd->vpage =
1753                     kmem_alloc(vpgtob(npages), KM_SLEEP);
1754                 for (i = 0; i < npages; i++) {
1755                         *nvp = *ovp++;
1756                         VPP_CLRPPLOCK(nvp++);
1757                 }
1758         } else
1759                 newsvd->vpage = NULL;

1761         /* Inform the vnode of the new mapping */
1762         if (newsvd->vp != NULL) {
1763                 error = VOP_ADDMAP(newsvd->vp, (offset_t)newsvd->offset,
1764                     newseg->s_as, newseg->s_base, newseg->s_size, newsvd->prot,
1765                     newsvd->maxprot, newsvd->type, newsvd->cred, NULL);
1766         }
1767 out:
1768         if (error == 0 && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1769                 ASSERT(newsvd->amp == NULL);
1770                 ASSERT(newsvd->tr_state == SEGVN_TR_OFF);
1771                 newsvd->rcookie = svd->rcookie;
1772                 hat_dup_region(newseg->s_as->a_hat, newsvd->rcookie);
1773         }
1774         return (error);
1775 }
______unchanged_portion_omitted_

1854 static int
1855 segvn_unmap(struct seg *seg, caddr_t addr, size_t len)
1856 {
1857         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1858         struct segvn_data *nsvd;
1859         struct seg *nseg;
1860         struct anon_map *amp;
1861         pgcnt_t opages;         /* old segment size in pages */
1862         pgcnt_t npages;         /* new segment size in pages */
1863         pgcnt_t dpages;         /* pages being deleted (unmapped) */
1864         hat_callback_t callback;        /* used for free_vp_pages() */
1865         hat_callback_t *cbp = NULL;
```

```
1866         caddr_t nbase;
1867         size_t nsize;
1868         size_t oswresv;
1869         int reclaim = 1;

1871         /*
1872          * We don't need any segment level locks for "segvn" data
1873          * since the address space is "write" locked.
1874          */
1875         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1875         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1877         /*
1878          * Fail the unmap if pages are SOFTLOCKed through this mapping.
1879          * softlockcnt is protected from change by the as write lock.
1880          */
1881 retry:
1882         if (svd->softlockcnt > 0) {
1883                 ASSERT(svd->tr_state == SEGVN_TR_OFF);

1885                 /*
1886                  * If this is shared segment non 0 softlockcnt
1887                  * means locked pages are still in use.
1888                  */
1889                 if (svd->type == MAP_SHARED) {
1890                         return (EAGAIN);
1891                 }

1893                 /*
1894                  * since we do have the writers lock nobody can fill
1895                  * the cache during the purge. The flush either succeeds
1896                  * or we still have pending I/Os.
1897                  */
1898                 if (reclaim == 1) {
1899                         segvn_purge(seg);
1900                         reclaim = 0;
1901                         goto retry;
1902                 }
1903                 return (EAGAIN);
1904         }

1906         /*
1907          * Check for bad sizes
1908          */
1909         if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
1910             (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET)) {
1911                 panic("segvn_unmap");
1912                 /*NOTREACHED*/
1913         }

1915         if (seg->s_szc != 0) {
1916                 size_t pgsz = page_get_pagesize(seg->s_szc);
1917                 int err;
1918                 if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
1919                         ASSERT(seg->s_base != addr || seg->s_size != len);
1920                         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1921                                 ASSERT(svd->amp == NULL);
1922                                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1923                                 hat_leave_region(seg->s_as->a_hat,
1924                                     svd->rcookie, HAT_REGION_TEXT);
1925                                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
1926                                 /*
1927                                  * could pass a flag to segvn_demote_range()
1928                                  * below to tell it not to do any unloads but
1929                                  * this case is rare enough to not bother for
1930                                  * now.
```



```
1931                                  */
1932                         } else if (svd->tr_state == SEGVN_TR_INIT) {
1933                                 svd->tr_state = SEGVN_TR_OFF;
1934                         } else if (svd->tr_state == SEGVN_TR_ON) {
1935                                 ASSERT(svd->amp != NULL);
1936                                 segvn_textunrepl(seg, 1);
1937                                 ASSERT(svd->amp == NULL);
1938                                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1939                         }
1940                         VM_STAT_ADD(segvnvmstats.demoterange[0]);
1941                         err = segvn_demote_range(seg, addr, len, SDR_END, 0);
1942                         if (err == 0) {
1943                                 return (IE_RETRY);
1944                         }
1945                         return (err);
1946                 }
1947         }

1949         /* Inform the vnode of the unmapping. */
1950         if (svd->vp) {
1951                 int error;

1953                 error = VOP_DELMAP(svd->vp,
1954                     (offset_t)svd->offset + (uintptr_t)(addr - seg->s_base),
1955                     seg->s_as, addr, len, svd->prot, svd->maxprot,
1956                     svd->type, svd->cred, NULL);

1958                 if (error == EAGAIN)
1959                         return (error);
1960         }

1962         /*
1963          * Remove any page locks set through this mapping.
1964          * If text replication is not off no page locks could have been
1965          * established via this mapping.
1966          */
1967         if (svd->tr_state == SEGVN_TR_OFF) {
1968                 (void) segvn_lockop(seg, addr, len, 0, MC_UNLOCK, NULL, 0);
1969         }

1971         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1972                 ASSERT(svd->amp == NULL);
1973                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1974                 ASSERT(svd->type == MAP_PRIVATE);
1975                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
1976                     HAT_REGION_TEXT);
1977                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
1978         } else if (svd->tr_state == SEGVN_TR_ON) {
1979                 ASSERT(svd->amp != NULL);
1980                 ASSERT(svd->pageprot == 0 && !(svd->prot & PROT_WRITE));
1981                 segvn_textunrepl(seg, 1);
1982                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
1983         } else {
1984                 if (svd->tr_state != SEGVN_TR_OFF) {
1985                         ASSERT(svd->tr_state == SEGVN_TR_INIT);
1986                         svd->tr_state = SEGVN_TR_OFF;
1987                 }
1988                 /*
1989                  * Unload any hardware translations in the range to be taken
1990                  * out. Use a callback to invoke free_vp_pages() effectively.
1991                  */
1992                 if (svd->vp != NULL && free_pages != 0) {
1993                         callback.hcb_data = seg;
1994                         callback.hcb_function = segvn_hat_unload_callback;
1995                         cbp = &callback;
1996                 }
```

```
1997                hat_unload_callback(seg->s_as->a_hat, addr, len,
1998                    HAT_UNLOAD_UNMAP, cbp);

2000                if (svd->type == MAP_SHARED && svd->vp != NULL &&
2001                    (svd->vp->v_flag & VVMEXEC) &&
2002                    ((svd->prot & PROT_WRITE) || svd->pageprot)) {
2003                        segvn_inval_trcache(svd->vp);
2004                }
2005        }

2007        /*
2008         * Check for entire segment
2009         */
2010        if (addr == seg->s_base && len == seg->s_size) {
2011                seg_free(seg);
2012                return (0);
2013        }

2015        opages = seg_pages(seg);
2016        dpages = btop(len);
2017        npages = opages - dpages;
2018        amp = svd->amp;
2019        ASSERT(amp == NULL || amp->a_szc >= seg->s_szc);

2021        /*
2022         * Check for beginning of segment
2023         */
2024        if (addr == seg->s_base) {
2025                if (svd->vpage != NULL) {
2026                        size_t nbytes;
2027                        struct vpage *ovpage;

2029                        ovpage = svd->vpage;    /* keep pointer to vpage */

2031                        nbytes = vpgtob(npages);
2032                        svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
2033                        bcopy(&ovpage[dpages], svd->vpage, nbytes);

2035                        /* free up old vpage */
2036                        kmem_free(ovpage, vpgtob(opages));
2037                }
2038                if (amp != NULL) {
2039                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2040                        if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2041                                /*
2042                                 * Shared anon map is no longer in use. Before
2043                                 * freeing its pages purge all entries from
2044                                 * pcache that belong to this amp.
2045                                 */
2046                                if (svd->type == MAP_SHARED) {
2047                                        ASSERT(amp->refcnt == 1);
2048                                        ASSERT(svd->softlockcnt == 0);
2049                                        anonmap_purge(amp);
2050                                }
2051                                /*
2052                                 * Free up now unused parts of anon_map array.
2053                                 */
2054                                if (amp->a_szc == seg->s_szc) {
2055                                        if (seg->s_szc != 0) {
2056                                                anon_free_pages(amp->ahp,
2057                                                    svd->anon_index, len,
2058                                                    seg->s_szc);
2059                                        } else {
2060                                                anon_free(amp->ahp,
2061                                                    svd->anon_index,
2062                                                    len);
```



```
2063                                        }
2064                                } else {
2065                                        ASSERT(svd->type == MAP_SHARED);
2066                                        ASSERT(amp->a_szc > seg->s_szc);
2067                                        anon_shmap_free_pages(amp,
2068                                            svd->anon_index, len);
2069                                }

2071                                /*
2072                                 * Unreserve swap space for the
2073                                 * unmapped chunk of this segment in
2074                                 * case it's MAP_SHARED
2075                                 */
2076                                if (svd->type == MAP_SHARED) {
2077                                        anon_unresv_zone(len,
2078                                            seg->s_as->a_proc->p_zone);
2079                                        amp->swresv -= len;
2080                                }
2081                        }
2082                        ANON_LOCK_EXIT(&amp->a_rwlock);
2083                        svd->anon_index += dpages;
2084                }
2085                if (svd->vp != NULL)
2086                        svd->offset += len;

2088                seg->s_base += len;
2089                seg->s_size -= len;

2091                if (svd->swresv) {
2092                        if (svd->flags & MAP_NORESERVE) {
2093                                ASSERT(amp);
2094                                oswresv = svd->swresv;

2096                                svd->swresv = ptob(anon_pages(amp->ahp,
2097                                    svd->anon_index, npages));
2098                                anon_unresv_zone(oswresv - svd->swresv,
2099                                    seg->s_as->a_proc->p_zone);
2100                                if (SEG_IS_PARTIAL_RESV(seg))
2101                                        seg->s_as->a_resvsize -= oswresv -
2102                                            svd->swresv;
2103                        } else {
2104                                size_t unlen;

2106                                if (svd->pageswap) {
2107                                        oswresv = svd->swresv;
2108                                        svd->swresv =
2109                                            segvn_count_swap_by_vpages(seg);
2110                                        ASSERT(oswresv >= svd->swresv);
2111                                        unlen = oswresv - svd->swresv;
2112                                } else {
2113                                        svd->swresv -= len;
2114                                        ASSERT(svd->swresv == seg->s_size);
2115                                        unlen = len;
2116                                }
2117                                anon_unresv_zone(unlen,
2118                                    seg->s_as->a_proc->p_zone);
2119                        }
2120                        TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2121                            seg, len, 0);
2122                }

2124                return (0);
2125        }

2127        /*
2128         * Check for end of segment
```

```
2129          */
2130         if (addr + len == seg->s_base + seg->s_size) {
2131                 if (svd->vpage != NULL) {
2132                         size_t nbytes;
2133                         struct vpage *ovpage;
2135                         ovpage = svd->vpage;    /* keep pointer to vpage */
2137                         nbytes = vpgtob(npages);
2138                         svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
2139                         bcopy(ovpage, svd->vpage, nbytes);
2141                         /* free up old vpage */
2142                         kmem_free(ovpage, vpgtob(opages));
2144                 }
2145                 if (amp != NULL) {
2146                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2147                         if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2148                                 /*
2149                                  * Free up now unused parts of anon_map array.
2150                                  */
2151                                 ulong_t an_idx = svd->anon_index + npages;
2153                                 /*
2154                                  * Shared anon map is no longer in use. Before
2155                                  * freeing its pages purge all entries from
2156                                  * pcache that belong to this amp.
2157                                  */
2158                                 if (svd->type == MAP_SHARED) {
2159                                         ASSERT(amp->refcnt == 1);
2160                                         ASSERT(svd->softlockcnt == 0);
2161                                         anonmap_purge(amp);
2162                                 }
2164                                 if (amp->a_szc == seg->s_szc) {
2165                                         if (seg->s_szc != 0) {
2166                                                 anon_free_pages(amp->ahp,
2167                                                     an_idx, len,
2168                                                     seg->s_szc);
2169                                         } else {
2170                                                 anon_free(amp->ahp, an_idx,
2171                                                     len);
2172                                         }
2173                                 } else {
2174                                         ASSERT(svd->type == MAP_SHARED);
2175                                         ASSERT(amp->a_szc > seg->s_szc);
2176                                         anon_shmap_free_pages(amp,
2177                                             an_idx, len);
2178                                 }
2180                                 /*
2181                                  * Unreserve swap space for the
2182                                  * unmapped chunk of this segment in
2183                                  * case it's MAP_SHARED
2184                                  */
2185                                 if (svd->type == MAP_SHARED) {
2186                                         anon_unresv_zone(len,
2187                                             seg->s_as->a_proc->p_zone);
2188                                         amp->swresv -= len;
2189                                 }
2190                         }
2191                         ANON_LOCK_EXIT(&amp->a_rwlock);
2192                 }
2194                 seg->s_size -= len;
```



```
2196                 if (svd->swresv) {
2197                         if (svd->flags & MAP_NORESERVE) {
2198                                 ASSERT(amp);
2199                                 oswresv = svd->swresv;
2200                                 svd->swresv = ptob(anon_pages(amp->ahp,
2201                                     svd->anon_index, npages));
2202                                 anon_unresv_zone(oswresv - svd->swresv,
2203                                     seg->s_as->a_proc->p_zone);
2204                                 if (SEG_IS_PARTIAL_RESV(seg))
2205                                         seg->s_as->a_resvsize -= oswresv -
2206                                             svd->swresv;
2207                         } else {
2208                                 size_t unlen;
2210                                 if (svd->pageswap) {
2211                                         oswresv = svd->swresv;
2212                                         svd->swresv =
2213                                             segvn_count_swap_by_vpages(seg);
2214                                         ASSERT(oswresv >= svd->swresv);
2215                                         unlen = oswresv - svd->swresv;
2216                                 } else {
2217                                         svd->swresv -= len;
2218                                         ASSERT(svd->swresv == seg->s_size);
2219                                         unlen = len;
2220                                 }
2221                                 anon_unresv_zone(unlen,
2222                                     seg->s_as->a_proc->p_zone);
2223                         }
2224                         TRACE_3(TR_FAC_VM, TR_ANON_PROC,
2225                             "anon proc:%p %lu %u", seg, len, 0);
2226                 }
2228                 return (0);
2229         }
2231         /*
2232          * The section to go is in the middle of the segment,
2233          * have to make it into two segments.  nseg is made for
2234          * the high end while seg is cut down at the low end.
2235          */
2236         nbase = addr + len;                             /* new seg base */
2237         nsize = (seg->s_base + seg->s_size) - nbase;    /* new seg size */
2238         seg->s_size = addr - seg->s_base;               /* shrink old seg */
2239         nseg = seg_alloc(seg->s_as, nbase, nsize);
2240         if (nseg == NULL) {
2241                 panic("segvn_unmap seg_alloc");
2242                 /*NOTREACHED*/
2243         }
2244         nseg->s_ops = seg->s_ops;
2245         nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
2246         nseg->s_data = (void *)nsvd;
2247         nseg->s_szc = seg->s_szc;
2248         *nsvd = *svd;
2249         nsvd->seg = nseg;
2250         nsvd->offset = svd->offset + (uintptr_t)(nseg->s_base - seg->s_base);
2251         nsvd->swresv = 0;
2252         nsvd->softlockcnt = 0;
2253         nsvd->softlockcnt_sbase = 0;
2254         nsvd->softlockcnt_send = 0;
2255         nsvd->svn_inz = svd->svn_inz;
2256         ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
2258         if (svd->vp != NULL) {
2259                 VN_HOLD(nsvd->vp);
2260                 if (nsvd->type == MAP_SHARED)
```

```
2261                        lgrp_shm_policy_init(NULL, nsvd->vp);
2262        }
2263        crhold(svd->cred);

2265        if (svd->vpage == NULL) {
2266                nsvd->vpage = NULL;
2267        } else {
2268                /* need to split vpage into two arrays */
2269                size_t nbytes;
2270                struct vpage *ovpage;

2272                ovpage = svd->vpage;            /* keep pointer to vpage */

2274                npages = seg_pages(seg);        /* seg has shrunk */
2275                nbytes = vpgtob(npages);
2276                svd->vpage = kmem_alloc(nbytes, KM_SLEEP);

2278                bcopy(ovpage, svd->vpage, nbytes);

2280                npages = seg_pages(nseg);
2281                nbytes = vpgtob(npages);
2282                nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);

2284                bcopy(&ovpage[opages - npages], nsvd->vpage, nbytes);

2286                /* free up old vpage */
2287                kmem_free(ovpage, vpgtob(opages));
2288        }

2290        if (amp == NULL) {
2291                nsvd->amp = NULL;
2292                nsvd->anon_index = 0;
2293        } else {
2294                /*
2295                 * Need to create a new anon map for the new segment.
2296                 * We'll also allocate a new smaller array for the old
2297                 * smaller segment to save space.
2298                 */
2299                opages = btop((uintptr_t)(addr - seg->s_base));
2300                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2301                if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2302                        /*
2303                         * Free up now unused parts of anon_map array.
2304                         */
2305                        ulong_t an_idx = svd->anon_index + opages;

2307                        /*
2308                         * Shared anon map is no longer in use. Before
2309                         * freeing its pages purge all entries from
2310                         * pcache that belong to this amp.
2311                         */
2312                        if (svd->type == MAP_SHARED) {
2313                                ASSERT(amp->refcnt == 1);
2314                                ASSERT(svd->softlockcnt == 0);
2315                                anonmap_purge(amp);
2316                        }

2318                        if (amp->a_szc == seg->s_szc) {
2319                                if (seg->s_szc != 0) {
2320                                        anon_free_pages(amp->ahp, an_idx, len,
2321                                            seg->s_szc);
2322                                } else {
2323                                        anon_free(amp->ahp, an_idx,
2324                                            len);
2325                                }
2326                        } else {
```



```
2327                                ASSERT(svd->type == MAP_SHARED);
2328                                ASSERT(amp->a_szc > seg->s_szc);
2329                                anon_shmap_free_pages(amp, an_idx, len);
2330                        }

2332                        /*
2333                         * Unreserve swap space for the
2334                         * unmapped chunk of this segment in
2335                         * case it's MAP_SHARED
2336                         */
2337                        if (svd->type == MAP_SHARED) {
2338                                anon_unresv_zone(len,
2339                                    seg->s_as->a_proc->p_zone);
2340                                amp->swresv -= len;
2341                        }
2342                }
2343                nsvd->anon_index = svd->anon_index +
2344                    btop((uintptr_t)(nseg->s_base - seg->s_base));
2345                if (svd->type == MAP_SHARED) {
2346                        amp->refcnt++;
2347                        nsvd->amp = amp;
2348                } else {
2349                        struct anon_map *namp;
2350                        struct anon_hdr *nahp;

2352                        ASSERT(svd->type == MAP_PRIVATE);
2353                        nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
2354                        namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
2355                        namp->a_szc = seg->s_szc;
2356                        (void) anon_copy_ptr(amp->ahp, svd->anon_index, nahp,
2357                            0, btop(seg->s_size), ANON_SLEEP);
2358                        (void) anon_copy_ptr(amp->ahp, nsvd->anon_index,
2359                            namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
2360                        anon_release(amp->ahp, btop(amp->size));
2361                        svd->anon_index = 0;
2362                        nsvd->anon_index = 0;
2363                        amp->ahp = nahp;
2364                        amp->size = seg->s_size;
2365                        nsvd->amp = namp;
2366                }
2367                ANON_LOCK_EXIT(&amp->a_rwlock);
2368        }
2369        if (svd->swresv) {
2370                if (svd->flags & MAP_NORESERVE) {
2371                        ASSERT(amp);
2372                        oswresv = svd->swresv;
2373                        svd->swresv = ptob(anon_pages(amp->ahp,
2374                            svd->anon_index, btop(seg->s_size)));
2375                        nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
2376                            nsvd->anon_index, btop(nseg->s_size)));
2377                        ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
2378                        anon_unresv_zone(oswresv - (svd->swresv + nsvd->swresv),
2379                            seg->s_as->a_proc->p_zone);
2380                        if (SEG_IS_PARTIAL_RESV(seg))
2381                                seg->s_as->a_resvsize -= oswresv -
2382                                    (svd->swresv + nsvd->swresv);
2383                } else {
2384                        size_t unlen;

2386                        if (svd->pageswap) {
2387                                oswresv = svd->swresv;
2388                                svd->swresv = segvn_count_swap_by_vpages(seg);
2389                                nsvd->swresv = segvn_count_swap_by_vpages(nseg);
2390                                ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
2391                                unlen = oswresv - (svd->swresv + nsvd->swresv);
2392                        } else {
```

```
2393                                if (seg->s_size + nseg->s_size + len !=
2394                                    svd->swresv) {
2395                                        panic("segvn_unmap: cannot split "
2396                                            "swap reservation");
2397                                        /*NOTREACHED*/
2398                                }
2399                                svd->swresv = seg->s_size;
2400                                nsvd->swresv = nseg->s_size;
2401                                unlen = len;
2402                        }
2403                        anon_unresv_zone(unlen,
2404                            seg->s_as->a_proc->p_zone);
2405                }
2406                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2407                    seg, len, 0);
2408        }

2410        return (0);                     /* I'm glad that's all over with! */
2411 }

2413 static void
2414 segvn_free(struct seg *seg)
2415 {
2416        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2417        pgcnt_t npages = seg_pages(seg);
2418        struct anon_map *amp;
2419        size_t len;

2421        /*
2422         * We don't need any segment level locks for "segvn" data
2423         * since the address space is "write" locked.
2424         */
2425        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
2425        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
2426        ASSERT(svd->tr_state == SEGVN_TR_OFF);

2428        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);

2430        /*
2431         * Be sure to unlock pages. XXX Why do things get free'ed instead
2432         * of unmapped? XXX
2433         */
2434        (void) segvn_lockop(seg, seg->s_base, seg->s_size,
2435            0, MC_UNLOCK, NULL, 0);

2437        /*
2438         * Deallocate the vpage and anon pointers if necessary and possible.
2439         */
2440        if (svd->vpage != NULL) {
2441                kmem_free(svd->vpage, vpgtob(npages));
2442                svd->vpage = NULL;
2443        }
2444        if ((amp = svd->amp) != NULL) {
2445                /*
2446                 * If there are no more references to this anon_map
2447                 * structure, then deallocate the structure after freeing
2448                 * up all the anon slot pointers that we can.
2449                 */
2450                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2451                ASSERT(amp->a_szc >= seg->s_szc);
2452                if (--amp->refcnt == 0) {
2453                        if (svd->type == MAP_PRIVATE) {
2454                                /*
2455                                 * Private - we only need to anon_free
2456                                 * the part that this segment refers to.
2457                                 */
```



```
2458                                if (seg->s_szc != 0) {
2459                                        anon_free_pages(amp->ahp,
2460                                            svd->anon_index, seg->s_size,
2461                                            seg->s_szc);
2462                                } else {
2463                                        anon_free(amp->ahp, svd->anon_index,
2464                                            seg->s_size);
2465                                }
2466                        } else {

2468                                /*
2469                                 * Shared anon map is no longer in use. Before
2470                                 * freeing its pages purge all entries from
2471                                 * pcache that belong to this amp.
2472                                 */
2473                                ASSERT(svd->softlockcnt == 0);
2474                                anonmap_purge(amp);

2476                                /*
2477                                 * Shared - anon_free the entire
2478                                 * anon_map's worth of stuff and
2479                                 * release any swap reservation.
2480                                 */
2481                                if (amp->a_szc != 0) {
2482                                        anon_shmap_free_pages(amp, 0,
2483                                            amp->size);
2484                                } else {
2485                                        anon_free(amp->ahp, 0, amp->size);
2486                                }
2487                                if ((len = amp->swresv) != 0) {
2488                                        anon_unresv_zone(len,
2489                                            seg->s_as->a_proc->p_zone);
2490                                        TRACE_3(TR_FAC_VM, TR_ANON_PROC,
2491                                            "anon proc:%p %lu %u", seg, len, 0);
2492                                }
2493                        }
2494                        svd->amp = NULL;
2495                        ANON_LOCK_EXIT(&amp->a_rwlock);
2496                        anonmap_free(amp);
2497                } else if (svd->type == MAP_PRIVATE) {
2498                        /*
2499                         * We had a private mapping which still has
2500                         * a held anon_map so just free up all the
2501                         * anon slot pointers that we were using.
2502                         */
2503                        if (seg->s_szc != 0) {
2504                                anon_free_pages(amp->ahp, svd->anon_index,
2505                                    seg->s_size, seg->s_szc);
2506                        } else {
2507                                anon_free(amp->ahp, svd->anon_index,
2508                                    seg->s_size);
2509                        }
2510                        ANON_LOCK_EXIT(&amp->a_rwlock);
2511                } else {
2512                        ANON_LOCK_EXIT(&amp->a_rwlock);
2513                }
2514        }

2516        /*
2517         * Release swap reservation.
2518         */
2519        if ((len = svd->swresv) != 0) {
2520                anon_unresv_zone(svd->swresv,
2521                    seg->s_as->a_proc->p_zone);
2522                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2523                    seg, len, 0);
```

```
2524                 if (SEG_IS_PARTIAL_RESV(seg))
2525                         seg->s_as->a_resvsize -= svd->swresv;
2526                 svd->swresv = 0;
2527         }
2528         /*
2529          * Release claim on vnode, credentials, and finally free the
2530          * private data.
2531          */
2532         if (svd->vp != NULL) {
2533                 if (svd->type == MAP_SHARED)
2534                         lgrp_shm_policy_fini(NULL, svd->vp);
2535                 VN_RELE(svd->vp);
2536                 svd->vp = NULL;
2537         }
2538         crfree(svd->cred);
2539         svd->pageprot = 0;
2540         svd->pageadvice = 0;
2541         svd->pageswap = 0;
2542         svd->cred = NULL;

2544         /*
2545          * Take segfree_syncmtx lock to let segvn_reclaim() finish if it's
2546          * still working with this segment without holding as lock (in case
2547          * it's called by pcache async thread).
2548          */
2549         ASSERT(svd->softlockcnt == 0);
2550         mutex_enter(&svd->segfree_syncmtx);
2551         mutex_exit(&svd->segfree_syncmtx);

2553         seg->s_data = NULL;
2554         kmem_cache_free(segvn_cache, svd);
2555 }

2557 /*
2558  * Do a F_SOFTUNLOCK call over the range requested.  The range must have
2559  * already been F_SOFTLOCK'ed.
2560  * Caller must always match addr and len of a softunlock with a previous
2561  * softlock with exactly the same addr and len.
2562  */
2563 static void
2564 segvn_softunlock(struct seg *seg, caddr_t addr, size_t len, enum seg_rw rw)
2565 {
2566         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2567         page_t *pp;
2568         caddr_t adr;
2569         struct vnode *vp;
2570         u_offset_t offset;
2571         ulong_t anon_index;
2572         struct anon_map *amp;
2573         struct anon *ap = NULL;

2575         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
2575         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
2576         ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

2578         if ((amp = svd->amp) != NULL)
2579                 anon_index = svd->anon_index + seg_page(seg, addr);

2581         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
2582                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
2583                 hat_unlock_region(seg->s_as->a_hat, addr, len, svd->rcookie);
2584         } else {
2585                 hat_unlock(seg->s_as->a_hat, addr, len);
2586         }
2587         for (adr = addr; adr < addr + len; adr += PAGESIZE) {
2588                 if (amp != NULL) {
```



```
2589                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2590                         if ((ap = anon_get_ptr(amp->ahp, anon_index++))
2591                             != NULL) {
2592                                 swap_xlate(ap, &vp, &offset);
2593                         } else {
2594                                 vp = svd->vp;
2595                                 offset = svd->offset +
2596                                     (uintptr_t)(adr - seg->s_base);
2597                         }
2598                         ANON_LOCK_EXIT(&amp->a_rwlock);
2599                 } else {
2600                         vp = svd->vp;
2601                         offset = svd->offset +
2602                             (uintptr_t)(adr - seg->s_base);
2603                 }

2605                 /*
2606                  * Use page_find() instead of page_lookup() to
2607                  * find the page since we know that it is locked.
2608                  */
2609                 pp = page_find(vp, offset);
2610                 if (pp == NULL) {
2611                         panic(
2612                             "segvn_softunlock: addr %p, ap %p, vp %p, off %llx",
2613                             (void *)adr, (void *)ap, (void *)vp, offset);
2614                         /*NOTREACHED*/
2615                 }

2617                 if (rw == S_WRITE) {
2618                         hat_setrefmod(pp);
2619                         if (seg->s_as->a_vbits)
2620                                 hat_setstat(seg->s_as, adr, PAGESIZE,
2621                                     P_REF | P_MOD);
2622                 } else if (rw != S_OTHER) {
2623                         hat_setref(pp);
2624                         if (seg->s_as->a_vbits)
2625                                 hat_setstat(seg->s_as, adr, PAGESIZE, P_REF);
2626                 }
2627                 TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
2628                     "segvn_fault:pp %p vp %p offset %llx", pp, vp, offset);
2629                 page_unlock(pp);
2630         }
2631         ASSERT(svd->softlockcnt >= btop(len));
2632         if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -btop(len))) {
2633                 /*
2634                  * All SOFTLOCKS are gone. Wakeup any waiting
2635                  * unmappers so they can try again to unmap.
2636                  * Check for waiters first without the mutex
2637                  * held so we don't always grab the mutex on
2638                  * softunlocks.
2639                  */
2640                 if (AS_ISUNMAPWAIT(seg->s_as)) {
2641                         mutex_enter(&seg->s_as->a_contents);
2642                         if (AS_ISUNMAPWAIT(seg->s_as)) {
2643                                 AS_CLRUNMAPWAIT(seg->s_as);
2644                                 cv_broadcast(&seg->s_as->a_cv);
2645                         }
2646                         mutex_exit(&seg->s_as->a_contents);
2647                 }
2648         }
2649 }
______unchanged_portion_omitted_

4919 int fltadvice = 1;      /* set to free behind pages for sequential access */

4921 /*
```

```
4922  * This routine is called via a machine specific fault handling routine.
4923  * It is also called by software routines wishing to lock or unlock
4924  * a range of addresses.
4925  *
4926  * Here is the basic algorithm:
4927  *      If unlocking
4928  *              Call segvn_softunlock
4929  *              Return
4930  *      endif
4931  *      Checking and set up work
4932  *      If we will need some non-anonymous pages
4933  *              Call VOP_GETPAGE over the range of non-anonymous pages
4934  *      endif
4935  *      Loop over all addresses requested
4936  *              Call segvn_faultpage passing in page list
4937  *                  to load up translations and handle anonymous pages
4938  *      endloop
4939  *      Load up translation to any additional pages in page list not
4940  *          already handled that fit into this segment
4941  */
4942 static faultcode_t
4943 segvn_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
4944     enum fault_type type, enum seg_rw rw)
4945 {
4946         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4947         page_t **plp, **ppp, *pp;
4948         u_offset_t off;
4949         caddr_t a;
4950         struct vpage *vpage;
4951         uint_t vpprot, prot;
4952         int err;
4953         page_t *pl[PVN_GETPAGE_NUM + 1];
4954         size_t plsz, pl_alloc_sz;
4955         size_t page;
4956         ulong_t anon_index;
4957         struct anon_map *amp;
4958         int dogetpage = 0;
4959         caddr_t lpgaddr, lpgeaddr;
4960         size_t pgsz;
4961         anon_sync_obj_t cookie;
4962         int brkcow = BREAK_COW_SHARE(rw, type, svd->type);

4964         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
4964         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
4965         ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);

4967         /*
4968          * First handle the easy stuff
4969          */
4970         if (type == F_SOFTUNLOCK) {
4971                 if (rw == S_READ_NOCOW) {
4972                         rw = S_READ;
4973                         ASSERT(AS_WRITE_HELD(seg->s_as));
4973                         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
4974                 }
4975                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4976                 pgsz = (seg->s_szc == 0) ? PAGESIZE :
4977                     page_get_pagesize(seg->s_szc);
4978                 VM_STAT_COND_ADD(pgsz > PAGESIZE, segvnvmstats.fltanpages[16]);
4979                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
4980                 segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
4981                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4982                 return (0);
4983         }

4985         ASSERT(svd->tr_state == SEGVN_TR_OFF ||
```



```
4986             !HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
4987         if (brkcow == 0) {
4988                 if (svd->tr_state == SEGVN_TR_INIT) {
4989                         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4990                         if (svd->tr_state == SEGVN_TR_INIT) {
4991                                 ASSERT(svd->vp != NULL && svd->amp == NULL);
4992                                 ASSERT(svd->flags & MAP_TEXT);
4993                                 ASSERT(svd->type == MAP_PRIVATE);
4994                                 segvn_textrepl(seg);
4995                                 ASSERT(svd->tr_state != SEGVN_TR_INIT);
4996                                 ASSERT(svd->tr_state != SEGVN_TR_ON ||
4997                                     svd->amp != NULL);
4998                         }
4999                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5000                 }
5001         } else if (svd->tr_state != SEGVN_TR_OFF) {
5002                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

5004                 if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
5005                         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
5006                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5007                         return (FC_PROT);
5008                 }

5010                 if (svd->tr_state == SEGVN_TR_ON) {
5011                         ASSERT(svd->vp != NULL && svd->amp != NULL);
5012                         segvn_textunrepl(seg, 0);
5013                         ASSERT(svd->amp == NULL &&
5014                             svd->tr_state == SEGVN_TR_OFF);
5015                 } else if (svd->tr_state != SEGVN_TR_OFF) {
5016                         svd->tr_state = SEGVN_TR_OFF;
5017                 }
5018                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
5019                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5020         }

5022 top:
5023         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

5025         /*
5026          * If we have the same protections for the entire segment,
5027          * insure that the access being attempted is legitimate.
5028          */

5030         if (svd->pageprot == 0) {
5031                 uint_t protchk;

5033                 switch (rw) {
5034                 case S_READ:
5035                 case S_READ_NOCOW:
5036                         protchk = PROT_READ;
5037                         break;
5038                 case S_WRITE:
5039                         protchk = PROT_WRITE;
5040                         break;
5041                 case S_EXEC:
5042                         protchk = PROT_EXEC;
5043                         break;
5044                 case S_OTHER:
5045                 default:
5046                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
5047                         break;
5048                 }

5050                 if ((svd->prot & protchk) == 0) {
5051                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
```

```
5052                        return (FC_PROT);       /* illegal access type */
5053                }
5054        }

5056        if (brkcow && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5057                /* this must be SOFTLOCK S_READ fault */
5058                ASSERT(svd->amp == NULL);
5059                ASSERT(svd->tr_state == SEGVN_TR_OFF);
5060                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5061                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5062                if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5063                        /*
5064                         * this must be the first ever non S_READ_NOCOW
5065                         * softlock for this segment.
5066                         */
5067                        ASSERT(svd->softlockcnt == 0);
5068                        hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5069                            HAT_REGION_TEXT);
5070                        svd->rcookie = HAT_INVALID_REGION_COOKIE;
5071                }
5072                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5073                goto top;
5074        }

5076        /*
5077         * We can't allow the long term use of softlocks for vmpss segments,
5078         * because in some file truncation cases we should be able to demote
5079         * the segment, which requires that there are no softlocks.  The
5080         * only case where it's ok to allow a SOFTLOCK fault against a vmpss
5081         * segment is S_READ_NOCOW, where the caller holds the address space
5082         * locked as writer and calls softunlock before dropping the as lock.
5083         * S_READ_NOCOW is used by /proc to read memory from another user.
5084         *
5085         * Another deadlock between SOFTLOCK and file truncation can happen
5086         * because segvn_fault_vnodepages() calls the FS one pagesize at
5087         * a time. A second VOP_GETPAGE() call by segvn_fault_vnodepages()
5088         * can cause a deadlock because the first set of page_t's remain
5089         * locked SE_SHARED.  To avoid this, we demote segments on a first
5090         * SOFTLOCK if they have a length greater than the segment's
5091         * page size.
5092         *
5093         * So for now, we only avoid demoting a segment on a SOFTLOCK when
5094         * the access type is S_READ_NOCOW and the fault length is less than
5095         * or equal to the segment's page size. While this is quite restrictive,
5096         * it should be the most common case of SOFTLOCK against a vmpss
5097         * segment.
5098         *
5099         * For S_READ_NOCOW, it's safe not to do a copy on write because the
5100         * caller makes sure no COW will be caused by another thread for a
5101         * softlocked page.
5102         */
5103        if (type == F_SOFTLOCK && svd->vp != NULL && seg->s_szc != 0) {
5104                int demote = 0;

5106                if (rw != S_READ_NOCOW) {
5107                        demote = 1;
5108                }
5109                if (!demote && len > PAGESIZE) {
5110                        pgsz = page_get_pagesize(seg->s_szc);
5111                        CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
5112                            lpgeaddr);
5113                        if (lpgeaddr - lpgaddr > pgsz) {
5114                                demote = 1;
5115                        }
5116                }
```



```
5118                ASSERT(demote || AS_WRITE_HELD(seg->s_as));
5118                ASSERT(demote || AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

5120                if (demote) {
5121                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5122                        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5123                        if (seg->s_szc != 0) {
5124                                segvn_vmpss_clrszc_cnt++;
5125                                ASSERT(svd->softlockcnt == 0);
5126                                err = segvn_clrszc(seg);
5127                                if (err) {
5128                                        segvn_vmpss_clrszc_err++;
5129                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5130                                        return (FC_MAKE_ERR(err));
5131                                }
5132                        }
5133                        ASSERT(seg->s_szc == 0);
5134                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5135                        goto top;
5136                }
5137        }

5139        /*
5140         * Check to see if we need to allocate an anon_map structure.
5141         */
5142        if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
5143                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5144                /*
5145                 * Drop the "read" lock on the segment and acquire
5146                 * the "write" version since we have to allocate the
5147                 * anon_map.
5148                 */
5149                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5150                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

5152                if (svd->amp == NULL) {
5153                        svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
5154                        svd->amp->a_szc = seg->s_szc;
5155                }
5156                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

5158                /*
5159                 * Start all over again since segment protections
5160                 * may have changed after we dropped the "read" lock.
5161                 */
5162                goto top;
5163        }

5165        /*
5166         * S_READ_NOCOW vs S_READ distinction was
5167         * only needed for the code above. After
5168         * that we treat it as S_READ.
5169         */
5170        if (rw == S_READ_NOCOW) {
5171                ASSERT(type == F_SOFTLOCK);
5172                ASSERT(AS_WRITE_HELD(seg->s_as));
5172                ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
5173                rw = S_READ;
5174        }

5176        amp = svd->amp;

5178        /*
5179         * MADV_SEQUENTIAL work is ignored for large page segments.
5180         */
5181        if (seg->s_szc != 0) {
```

```
5182                 pgsz = page_get_pagesize(seg->s_szc);
5183                 ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
5184                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
5185                 if (svd->vp == NULL) {
5186                         err = segvn_fault_anonpages(hat, seg, lpgaddr,
5187                             lpgeaddr, type, rw, addr, addr + len, brkcow);
5188                 } else {
5189                         err = segvn_fault_vnodepages(hat, seg, lpgaddr,
5190                             lpgeaddr, type, rw, addr, addr + len, brkcow);
5191                         if (err == IE_RETRY) {
5192                                 ASSERT(seg->s_szc == 0);
5193                                 ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
5194                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5195                                 goto top;
5196                         }
5197                 }
5198                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5199                 return (err);
5200         }

5202         page = seg_page(seg, addr);
5203         if (amp != NULL) {
5204                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5205                 anon_index = svd->anon_index + page;

5207                 if (type == F_PROT && rw == S_READ &&
5208                     svd->tr_state == SEGVN_TR_OFF &&
5209                     svd->type == MAP_PRIVATE && svd->pageprot == 0) {
5210                         size_t index = anon_index;
5211                         struct anon *ap;

5213                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5214                         /*
5215                          * The fast path could apply to S_WRITE also, except
5216                          * that the protection fault could be caused by lazy
5217                          * tlb flush when ro->rw. In this case, the pte is
5218                          * RW already. But RO in the other cpu's tlb causes
5219                          * the fault. Since hat_chgprot won't do anything if
5220                          * pte doesn't change, we may end up faulting
5221                          * indefinitely until the RO tlb entry gets replaced.
5222                          */
5223                         for (a = addr; a < addr + len; a += PAGESIZE, index++) {
5224                                 anon_array_enter(amp, index, &cookie);
5225                                 ap = anon_get_ptr(amp->ahp, index);
5226                                 anon_array_exit(&cookie);
5227                                 if ((ap == NULL) || (ap->an_refcnt != 1)) {
5228                                         ANON_LOCK_EXIT(&amp->a_rwlock);
5229                                         goto slow;
5230                                 }
5231                         }
5232                         hat_chgprot(seg->s_as->a_hat, addr, len, svd->prot);
5233                         ANON_LOCK_EXIT(&amp->a_rwlock);
5234                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5235                         return (0);
5236                 }
5237         }
5238 slow:

5240         if (svd->vpage == NULL)
5241                 vpage = NULL;
5242         else
5243                 vpage = &svd->vpage[page];

5245         off = svd->offset + (uintptr_t)(addr - seg->s_base);

5247         /*
```



```
5248          * If MADV_SEQUENTIAL has been set for the particular page we
5249          * are faulting on, free behind all pages in the segment and put
5250          * them on the free list.
5251          */

5253         if ((page != 0) && fltadvice && svd->tr_state != SEGVN_TR_ON) {
5254                 struct vpage *vpp;
5255                 ulong_t fanon_index;
5256                 size_t fpage;
5257                 u_offset_t pgoff, fpgoff;
5258                 struct vnode *fvp;
5259                 struct anon *fap = NULL;

5261                 if (svd->advice == MADV_SEQUENTIAL ||
5262                     (svd->pageadvice &&
5263                     VPP_ADVICE(vpage) == MADV_SEQUENTIAL)) {
5264                         pgoff = off - PAGESIZE;
5265                         fpage = page - 1;
5266                         if (vpage != NULL)
5267                                 vpp = &svd->vpage[fpage];
5268                         if (amp != NULL)
5269                                 fanon_index = svd->anon_index + fpage;

5271                         while (pgoff > svd->offset) {
5272                                 if (svd->advice != MADV_SEQUENTIAL &&
5273                                     (!svd->pageadvice || (vpage &&
5274                                     VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
5275                                         break;

5277                                 /*
5278                                  * If this is an anon page, we must find the
5279                                  * correct <vp, offset> for it
5280                                  */
5281                                 fap = NULL;
5282                                 if (amp != NULL) {
5283                                         ANON_LOCK_ENTER(&amp->a_rwlock,
5284                                             RW_READER);
5285                                         anon_array_enter(amp, fanon_index,
5286                                             &cookie);
5287                                         fap = anon_get_ptr(amp->ahp,
5288                                             fanon_index);
5289                                         if (fap != NULL) {
5290                                                 swap_xlate(fap, &fvp, &fpgoff);
5291                                         } else {
5292                                                 fpgoff = pgoff;
5293                                                 fvp = svd->vp;
5294                                         }
5295                                         anon_array_exit(&cookie);
5296                                         ANON_LOCK_EXIT(&amp->a_rwlock);
5297                                 } else {
5298                                         fpgoff = pgoff;
5299                                         fvp = svd->vp;
5300                                 }
5301                                 if (fvp == NULL)
5302                                         break;  /* XXX */
5303                                 /*
5304                                  * Skip pages that are free or have an
5305                                  * "exclusive" lock.
5306                                  */
5307                                 pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
5308                                 if (pp == NULL)
5309                                         break;
5310                                 /*
5311                                  * We don't need the page_struct_lock to test
5312                                  * as this is only advisory; even if we
5313                                  * acquire it someone might race in and lock
```

```
5314                                 * the page after we unlock and before the
5315                                 * PUTPAGE, then VOP_PUTPAGE will do nothing.
5316                                 */
5317                                if (pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
5318                                        /*
5319                                         * Hold the vnode before releasing
5320                                         * the page lock to prevent it from
5321                                         * being freed and re-used by some
5322                                         * other thread.
5323                                         */
5324                                        VN_HOLD(fvp);
5325                                        page_unlock(pp);
5326                                        /*
5327                                         * We should build a page list
5328                                         * to kluster putpages XXX
5329                                         */
5330                                        (void) VOP_PUTPAGE(fvp,
5331                                            (offset_t)fpgoff, PAGESIZE,
5332                                            (B_DONTNEED|B_FREE|B_ASYNC),
5333                                            svd->cred, NULL);
5334                                        VN_RELE(fvp);
5335                                } else {
5336                                        /*
5337                                         * XXX - Should the loop terminate if
5338                                         * the page is 'locked'?
5339                                         */
5340                                        page_unlock(pp);
5341                                }
5342                                --vpp;
5343                                --fanon_index;
5344                                pgoff -= PAGESIZE;
5345                        }
5346                }
5347        }

5349        plp = pl;
5350        *plp = NULL;
5351        pl_alloc_sz = 0;

5353        /*
5354         * See if we need to call VOP_GETPAGE for
5355         * *any* of the range being faulted on.
5356         * We can skip all of this work if there
5357         * was no original vnode.
5358         */
5359        if (svd->vp != NULL) {
5360                u_offset_t vp_off;
5361                size_t vp_len;
5362                struct anon *ap;
5363                vnode_t *vp;

5365                vp_off = off;
5366                vp_len = len;

5368                if (amp == NULL)
5369                        dogetpage = 1;
5370                else {
5371                        /*
5372                         * Only acquire reader lock to prevent amp->ahp
5373                         * from being changed.  It's ok to miss pages,
5374                         * hence we don't do anon_array_enter
5375                         */
5376                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5377                        ap = anon_get_ptr(amp->ahp, anon_index);

5379                        if (len <= PAGESIZE)
```



```
5380                                /* inline non_anon() */
5381                                dogetpage = (ap == NULL);
5382                        else
5383                                dogetpage = non_anon(amp->ahp, anon_index,
5384                                    &vp_off, &vp_len);
5385                        ANON_LOCK_EXIT(&amp->a_rwlock);
5386                }

5388                if (dogetpage) {
5389                        enum seg_rw arw;
5390                        struct as *as = seg->s_as;

5392                        if (len > ptob((sizeof (pl) / sizeof (pl[0])) - 1)) {
5393                                /*
5394                                 * Page list won't fit in local array,
5395                                 * allocate one of the needed size.
5396                                 */
5397                                pl_alloc_sz =
5398                                    (btop(len) + 1) * sizeof (page_t *);
5399                                plp = kmem_alloc(pl_alloc_sz, KM_SLEEP);
5400                                plp[0] = NULL;
5401                                plsz = len;
5402                        } else if (rw == S_WRITE && svd->type == MAP_PRIVATE ||
5403                            svd->tr_state == SEGVN_TR_ON || rw == S_OTHER ||
5404                            (((size_t)(addr + PAGESIZE) <
5405                            (size_t)(seg->s_base + seg->s_size)) &&
5406                            hat_probe(as->a_hat, addr + PAGESIZE))) {
5407                                /*
5408                                 * Ask VOP_GETPAGE to return the exact number
5409                                 * of pages if
5410                                 * (a) this is a COW fault, or
5411                                 * (b) this is a software fault, or
5412                                 * (c) next page is already mapped.
5413                                 */
5414                                plsz = len;
5415                        } else {
5416                                /*
5417                                 * Ask VOP_GETPAGE to return adjacent pages
5418                                 * within the segment.
5419                                 */
5420                                plsz = MIN((size_t)PVN_GETPAGE_SZ, (size_t)
5421                                    ((seg->s_base + seg->s_size) - addr));
5422                                ASSERT((addr + plsz) <=
5423                                    (seg->s_base + seg->s_size));
5424                        }

5426                        /*
5427                         * Need to get some non-anonymous pages.
5428                         * We need to make only one call to GETPAGE to do
5429                         * this to prevent certain deadlocking conditions
5430                         * when we are doing locking.  In this case
5431                         * non_anon() should have picked up the smallest
5432                         * range which includes all the non-anonymous
5433                         * pages in the requested range.  We have to
5434                         * be careful regarding which rw flag to pass in
5435                         * because on a private mapping, the underlying
5436                         * object is never allowed to be written.
5437                         */
5438                        if (rw == S_WRITE && svd->type == MAP_PRIVATE) {
5439                                arw = S_READ;
5440                        } else {
5441                                arw = rw;
5442                        }
5443                        vp = svd->vp;
5444                        TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
5445                            "segvn_getpage:seg %p addr %p vp %p",
```

```
5446                            seg, addr, vp);
5447                        err = VOP_GETPAGE(vp, (offset_t)vp_off, vp_len,
5448                            &vpprot, plp, plsz, seg, addr + (vp_off - off), arw,
5449                            svd->cred, NULL);
5450                        if (err) {
5451                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5452                                segvn_pagelist_rele(plp);
5453                                if (pl_alloc_sz)
5454                                        kmem_free(plp, pl_alloc_sz);
5455                                return (FC_MAKE_ERR(err));
5456                        }
5457                        if (svd->type == MAP_PRIVATE)
5458                                vpprot &= ~PROT_WRITE;
5459                }
5460        }

5462        /*
5463         * N.B. at this time the plp array has all the needed non-anon
5464         * pages in addition to (possibly) having some adjacent pages.
5465         */

5467        /*
5468         * Always acquire the anon_array_lock to prevent
5469         * 2 threads from allocating separate anon slots for
5470         * the same "addr".
5471         *
5472         * If this is a copy-on-write fault and we don't already
5473         * have the anon_array_lock, acquire it to prevent the
5474         * fault routine from handling multiple copy-on-write faults
5475         * on the same "addr" in the same address space.
5476         *
5477         * Only one thread should deal with the fault since after
5478         * it is handled, the other threads can acquire a translation
5479         * to the newly created private page.  This prevents two or
5480         * more threads from creating different private pages for the
5481         * same fault.
5482         *
5483         * We grab "serialization" lock here if this is a MAP_PRIVATE segment
5484         * to prevent deadlock between this thread and another thread
5485         * which has soft-locked this page and wants to acquire serial_lock.
5486         * ( bug 4026339 )
5487         *
5488         * The fix for bug 4026339 becomes unnecessary when using the
5489         * locking scheme with per amp rwlock and a global set of hash
5490         * lock, anon_array_lock.  If we steal a vnode page when low
5491         * on memory and upgrad the page lock through page_rename,
5492         * then the page is PAGE_HANDLED, nothing needs to be done
5493         * for this page after returning from segvn_faultpage.
5494         *
5495         * But really, the page lock should be downgraded after
5496         * the stolen page is page_rename'd.
5497         */

5499        if (amp != NULL)
5500                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

5502        /*
5503         * Ok, now loop over the address range and handle faults
5504         */
5505        for (a = addr; a < addr + len; a += PAGESIZE, off += PAGESIZE) {
5506                err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
5507                    type, rw, brkcow);
5508                if (err) {
5509                        if (amp != NULL)
5510                                ANON_LOCK_EXIT(&amp->a_rwlock);
5511                        if (type == F_SOFTLOCK && a > addr) {
```



```
5512                                segvn_softunlock(seg, addr, (a - addr),
5513                                    S_OTHER);
5514                        }
5515                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5516                        segvn_pagelist_rele(plp);
5517                        if (pl_alloc_sz)
5518                                kmem_free(plp, pl_alloc_sz);
5519                        return (err);
5520                }
5521                if (vpage) {
5522                        vpage++;
5523                } else if (svd->vpage) {
5524                        page = seg_page(seg, addr);
5525                        vpage = &svd->vpage[++page];
5526                }
5527        }

5529        /* Didn't get pages from the underlying fs so we're done */
5530        if (!dogetpage)
5531                goto done;

5533        /*
5534         * Now handle any other pages in the list returned.
5535         * If the page can be used, load up the translations now.
5536         * Note that the for loop will only be entered if "plp"
5537         * is pointing to a non-NULL page pointer which means that
5538         * VOP_GETPAGE() was called and vpprot has been initialized.
5539         */
5540        if (svd->pageprot == 0)
5541                prot = svd->prot & vpprot;


5544        /*
5545         * Large Files: diff should be unsigned value because we started
5546         * supporting > 2GB segment sizes from 2.5.1 and when a
5547         * large file of size > 2GB gets mapped to address space
5548         * the diff value can be > 2GB.
5549         */

5551        for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
5552                size_t diff;
5553                struct anon *ap;
5554                int anon_index;
5555                anon_sync_obj_t cookie;
5556                int hat_flag = HAT_LOAD_ADV;

5558                if (svd->flags & MAP_TEXT) {
5559                        hat_flag |= HAT_LOAD_TEXT;
5560                }

5562                if (pp == PAGE_HANDLED)
5563                        continue;

5565                if (svd->tr_state != SEGVN_TR_ON &&
5566                    pp->p_offset >=  svd->offset &&
5567                    pp->p_offset < svd->offset + seg->s_size) {

5569                        diff = pp->p_offset - svd->offset;

5571                        /*
5572                         * Large Files: Following is the assertion
5573                         * validating the above cast.
5574                         */
5575                        ASSERT(svd->vp == pp->p_vnode);

5577                        page = btop(diff);
```

```
5578                        if (svd->pageprot)
5579                                prot = VPP_PROT(&svd->vpage[page]) & vpprot;
5580
5581                        /*
5582                         * Prevent other threads in the address space from
5583                         * creating private pages (i.e., allocating anon slots)
5584                         * while we are in the process of loading translations
5585                         * to additional pages returned by the underlying
5586                         * object.
5587                         */
5588                        if (amp != NULL) {
5589                                anon_index = svd->anon_index + page;
5590                                anon_array_enter(amp, anon_index, &cookie);
5591                                ap = anon_get_ptr(amp->ahp, anon_index);
5592                        }
5593                        if ((amp == NULL) || (ap == NULL)) {
5594                                if (IS_VMODSORT(pp->p_vnode) ||
5595                                    enable_mbit_wa) {
5596                                        if (rw == S_WRITE)
5597                                                hat_setmod(pp);
5598                                        else if (rw != S_OTHER &&
5599                                            !hat_ismod(pp))
5600                                                prot &= ~PROT_WRITE;
5601                                }
5602                                /*
5603                                 * Skip mapping read ahead pages marked
5604                                 * for migration, so they will get migrated
5605                                 * properly on fault
5606                                 */
5607                                ASSERT(amp == NULL ||
5608                                    svd->rcookie == HAT_INVALID_REGION_COOKIE);
5609                                if ((prot & PROT_READ) && !PP_ISMIGRATE(pp)) {
5610                                        hat_memload_region(hat,
5611                                            seg->s_base + diff,
5612                                            pp, prot, hat_flag,
5613                                            svd->rcookie);
5614                                }
5615                        }
5616                        if (amp != NULL)
5617                                anon_array_exit(&cookie);
5618                }
5619                page_unlock(pp);
5620        }
5621 done:
5622        if (amp != NULL)
5623                ANON_LOCK_EXIT(&amp->a_rwlock);
5624        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5625        if (pl_alloc_sz)
5626                kmem_free(plp, pl_alloc_sz);
5627        return (0);
5628 }
5629
5630 /*
5631  * This routine is used to start I/O on pages asynchronously.  XXX it will
5632  * only create PAGESIZE pages. At fault time they will be relocated into
5633  * larger pages.
5634  */
5635 static faultcode_t
5636 segvn_faulta(struct seg *seg, caddr_t addr)
5637 {
5638        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
5639        int err;
5640        struct anon_map *amp;
5641        vnode_t *vp;
5642
5643        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
```



```
5643        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
5644
5645        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
5646        if ((amp = svd->amp) != NULL) {
5647                struct anon *ap;
5648
5649                /*
5650                 * Reader lock to prevent amp->ahp from being changed.
5651                 * This is advisory, it's ok to miss a page, so
5652                 * we don't do anon_array_enter lock.
5653                 */
5654                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5655                if ((ap = anon_get_ptr(amp->ahp,
5656                    svd->anon_index + seg_page(seg, addr))) != NULL) {
5657
5658                        err = anon_getpage(&ap, NULL, NULL,
5659                            0, seg, addr, S_READ, svd->cred);
5660
5661                        ANON_LOCK_EXIT(&amp->a_rwlock);
5662                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5663                        if (err)
5664                                return (FC_MAKE_ERR(err));
5665                        return (0);
5666                }
5667                ANON_LOCK_EXIT(&amp->a_rwlock);
5668        }
5669
5670        if (svd->vp == NULL) {
5671                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5672                return (0);                     /* zfod page - do nothing now */
5673        }
5674
5675        vp = svd->vp;
5676        TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
5677            "segvn_getpage:seg %p addr %p vp %p", seg, addr, vp);
5678        err = VOP_GETPAGE(vp,
5679            (offset_t)(svd->offset + (uintptr_t)(addr - seg->s_base)),
5680            PAGESIZE, NULL, NULL, 0, seg, addr,
5681            S_OTHER, svd->cred, NULL);
5682
5683        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5684        if (err)
5685                return (FC_MAKE_ERR(err));
5686        return (0);
5687 }
5688
5689 static int
5690 segvn_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
5691 {
5692        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
5693        struct vpage *cvp, *svp, *evp;
5694        struct vnode *vp;
5695        size_t pgsz;
5696        pgcnt_t pgcnt;
5697        anon_sync_obj_t cookie;
5698        int unload_done = 0;
5699
5700        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
5700        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
5701
5702        if ((svd->maxprot & prot) != prot)
5703                return (EACCES);                        /* violated maxprot */
5704
5705        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5706
5707        /* return if prot is the same */
```

```
5708         if (!svd->pageprot && svd->prot == prot) {
5709                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5710                 return (0);
5711         }
5712
5713         /*
5714          * Since we change protections we first have to flush the cache.
5715          * This makes sure all the pagelock calls have to recheck
5716          * protections.
5717          */
5718         if (svd->softlockcnt > 0) {
5719                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
5720
5721                 /*
5722                  * If this is shared segment non 0 softlockcnt
5723                  * means locked pages are still in use.
5724                  */
5725                 if (svd->type == MAP_SHARED) {
5726                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5727                         return (EAGAIN);
5728                 }
5729
5730                 /*
5731                  * Since we do have the segvn writers lock nobody can fill
5732                  * the cache with entries belonging to this seg during
5733                  * the purge. The flush either succeeds or we still have
5734                  * pending I/Os.
5735                  */
5736                 segvn_purge(seg);
5737                 if (svd->softlockcnt > 0) {
5738                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5739                         return (EAGAIN);
5740                 }
5741         }
5742
5743         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5744                 ASSERT(svd->amp == NULL);
5745                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
5746                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5747                     HAT_REGION_TEXT);
5748                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
5749                 unload_done = 1;
5750         } else if (svd->tr_state == SEGVN_TR_INIT) {
5751                 svd->tr_state = SEGVN_TR_OFF;
5752         } else if (svd->tr_state == SEGVN_TR_ON) {
5753                 ASSERT(svd->amp != NULL);
5754                 segvn_textunrepl(seg, 0);
5755                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
5756                 unload_done = 1;
5757         }
5758
5759         if ((prot & PROT_WRITE) && svd->type == MAP_SHARED &&
5760             svd->vp != NULL && (svd->vp->v_flag & VVMEXEC)) {
5761                 ASSERT(vn_is_mapped(svd->vp, V_WRITE));
5762                 segvn_inval_trcache(svd->vp);
5763         }
5764         if (seg->s_szc != 0) {
5765                 int err;
5766                 pgsz = page_get_pagesize(seg->s_szc);
5767                 pgcnt = pgsz >> PAGESHIFT;
5768                 ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
5769                 if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
5770                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5771                         ASSERT(seg->s_base != addr || seg->s_size != len);
5772                         /*
5773                          * If we are holding the as lock as a reader then
```



```
5774                          * we need to return IE_RETRY and let the as
5775                          * layer drop and re-acquire the lock as a writer.
5776                          */
5777                         if (AS_READ_HELD(seg->s_as))
5777                         if (AS_READ_HELD(seg->s_as, &seg->s_as->a_lock))
5778                                 return (IE_RETRY);
5779                         VM_STAT_ADD(segvnvmstats.demoterange[1]);
5780                         if (svd->type == MAP_PRIVATE || svd->vp != NULL) {
5781                                 err = segvn_demote_range(seg, addr, len,
5782                                     SDR_END, 0);
5783                         } else {
5784                                 uint_t szcvec = map_pgszcvec(seg->s_base,
5785                                     pgsz, (uintptr_t)seg->s_base,
5786                                     (svd->flags & MAP_TEXT), MAPPGSZC_SHM, 0);
5787                                 err = segvn_demote_range(seg, addr, len,
5788                                     SDR_END, szcvec);
5789                         }
5790                         if (err == 0)
5791                                 return (IE_RETRY);
5792                         if (err == ENOMEM)
5793                                 return (IE_NOMEM);
5794                         return (err);
5795                 }
5796         }


5799         /*
5800          * If it's a private mapping and we're making it writable then we
5801          * may have to reserve the additional swap space now. If we are
5802          * making writable only a part of the segment then we use its vpage
5803          * array to keep a record of the pages for which we have reserved
5804          * swap. In this case we set the pageswap field in the segment's
5805          * segvn structure to record this.
5806          *
5807          * If it's a private mapping to a file (i.e., vp != NULL) and we're
5808          * removing write permission on the entire segment and we haven't
5809          * modified any pages, we can release the swap space.
5810          */
5811         if (svd->type == MAP_PRIVATE) {
5812                 if (prot & PROT_WRITE) {
5813                         if (!(svd->flags & MAP_NORESERVE) &&
5814                             !(svd->swresv && svd->pageswap == 0)) {
5815                                 size_t sz = 0;
5816
5817                                 /*
5818                                  * Start by determining how much swap
5819                                  * space is required.
5820                                  */
5821                                 if (addr == seg->s_base &&
5822                                     len == seg->s_size &&
5823                                     svd->pageswap == 0) {
5824                                         /* The whole segment */
5825                                         sz = seg->s_size;
5826                                 } else {
5827                                         /*
5828                                          * Make sure that the vpage array
5829                                          * exists, and make a note of the
5830                                          * range of elements corresponding
5831                                          * to len.
5832                                          */
5833                                         segvn_vpage(seg);
5834                                         if (svd->vpage == NULL) {
5835                                                 SEGVN_LOCK_EXIT(seg->s_as,
5836                                                     &svd->lock);
5837                                                 return (ENOMEM);
5838                                         }
```

```
5839                                    svp = &svd->vpage[seg_page(seg, addr)];
5840                                    evp = &svd->vpage[seg_page(seg,
5841                                        addr + len)];

5843                                    if (svd->pageswap == 0) {
5844                                            /*
5845                                             * This is the first time we've
5846                                             * asked for a part of this
5847                                             * segment, so we need to
5848                                             * reserve everything we've
5849                                             * been asked for.
5850                                             */
5851                                            sz = len;
5852                                    } else {
5853                                            /*
5854                                             * We have to count the number
5855                                             * of pages required.
5856                                             */
5857                                            for (cvp = svp;  cvp < evp;
5858                                                cvp++) {
5859                                                    if (!VPP_ISSWAPRES(cvp))
5860                                                            sz++;
5861                                            }
5862                                            sz <<= PAGESHIFT;
5863                                    }
5864                            }

5866                            /* Try to reserve the necessary swap. */
5867                            if (anon_resv_zone(sz,
5868                                seg->s_as->a_proc->p_zone) == 0) {
5869                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5870                                return (IE_NOMEM);
5871                            }

5873                            /*
5874                             * Make a note of how much swap space
5875                             * we've reserved.
5876                             */
5877                            if (svd->pageswap == 0 && sz == seg->s_size) {
5878                                    svd->swresv = sz;
5879                            } else {
5880                                    ASSERT(svd->vpage != NULL);
5881                                    svd->swresv += sz;
5882                                    svd->pageswap = 1;
5883                                    for (cvp = svp; cvp < evp; cvp++) {
5884                                            if (!VPP_ISSWAPRES(cvp))
5885                                                    VPP_SETSWAPRES(cvp);
5886                                    }
5887                            }
5888                    }
5889            } else {
5890                    /*
5891                     * Swap space is released only if this segment
5892                     * does not map anonymous memory, since read faults
5893                     * on such segments still need an anon slot to read
5894                     * in the data.
5895                     */
5896                    if (svd->swresv != 0 && svd->vp != NULL &&
5897                        svd->amp == NULL && addr == seg->s_base &&
5898                        len == seg->s_size && svd->pageprot == 0) {
5899                            ASSERT(svd->pageswap == 0);
5900                            anon_unresv_zone(svd->swresv,
5901                                seg->s_as->a_proc->p_zone);
5902                            svd->swresv = 0;
5903                            TRACE_3(TR_FAC_VM, TR_ANON_PROC,
5904                                "anon proc:%p %lu %u", seg, 0, 0);
```



```
5905                            }
5906                    }
5907            }

5909            if (addr == seg->s_base && len == seg->s_size && svd->vpage == NULL) {
5910                    if (svd->prot == prot) {
5911                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5912                            return (0);                     /* all done */
5913                    }
5914                    svd->prot = (uchar_t)prot;
5915            } else if (svd->type == MAP_PRIVATE) {
5916                    struct anon *ap = NULL;
5917                    page_t *pp;
5918                    u_offset_t offset, off;
5919                    struct anon_map *amp;
5920                    ulong_t anon_idx = 0;

5922                    /*
5923                     * A vpage structure exists or else the change does not
5924                     * involve the entire segment.  Establish a vpage structure
5925                     * if none is there.  Then, for each page in the range,
5926                     * adjust its individual permissions.  Note that write-
5927                     * enabling a MAP_PRIVATE page can affect the claims for
5928                     * locked down memory.  Overcommitting memory terminates
5929                     * the operation.
5930                     */
5931                    segvn_vpage(seg);
5932                    if (svd->vpage == NULL) {
5933                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5934                            return (ENOMEM);
5935                    }
5936                    svd->pageprot = 1;
5937                    if ((amp = svd->amp) != NULL) {
5938                            anon_idx = svd->anon_index + seg_page(seg, addr);
5939                            ASSERT(seg->s_szc == 0 ||
5940                                IS_P2ALIGNED(anon_idx, pgcnt));
5941                            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5942                    }

5944                    offset = svd->offset + (uintptr_t)(addr - seg->s_base);
5945                    evp = &svd->vpage[seg_page(seg, addr + len)];

5947                    /*
5948                     * See Statement at the beginning of segvn_lockop regarding
5949                     * the way cowcnts and lckcnts are handled.
5950                     */
5951                    for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {

5953                            if (seg->s_szc != 0) {
5954                                    if (amp != NULL) {
5955                                            anon_array_enter(amp, anon_idx,
5956                                                &cookie);
5957                                    }
5958                                    if (IS_P2ALIGNED(anon_idx, pgcnt) &&
5959                                        !segvn_claim_pages(seg, svp, offset,
5960                                        anon_idx, prot)) {
5961                                            if (amp != NULL) {
5962                                                    anon_array_exit(&cookie);
5963                                            }
5964                                            break;
5965                                    }
5966                                    if (amp != NULL) {
5967                                            anon_array_exit(&cookie);
5968                                    }
5969                                    anon_idx++;
5970                            } else {
```

```
5971                                if (amp != NULL) {
5972                                        anon_array_enter(amp, anon_idx,
5973                                            &cookie);
5974                                        ap = anon_get_ptr(amp->ahp, anon_idx++);
5975                                }

5977                                if (VPP_ISPPLOCK(svp) &&
5978                                    VPP_PROT(svp) != prot) {

5980                                        if (amp == NULL || ap == NULL) {
5981                                                vp = svd->vp;
5982                                                off = offset;
5983                                        } else
5984                                                swap_xlate(ap, &vp, &off);
5985                                        if (amp != NULL)
5986                                                anon_array_exit(&cookie);

5988                                        if ((pp = page_lookup(vp, off,
5989                                            SE_SHARED)) == NULL) {
5990                                                panic("segvn_setprot: no page");
5991                                                /*NOTREACHED*/
5992                                        }
5993                                        ASSERT(seg->s_szc == 0);
5994                                        if ((VPP_PROT(svp) ^ prot) &
5995                                            PROT_WRITE) {
5996                                                if (prot & PROT_WRITE) {
5997                                                        if (!page_addclaim(
5998                                                            pp)) {
5999                                                                page_unlock(pp);
6000                                                                break;
6001                                                        }
6002                                                } else {
6003                                                        if (!page_subclaim(
6004                                                            pp)) {
6005                                                                page_unlock(pp);
6006                                                                break;
6007                                                        }
6008                                                }
6009                                        }
6010                                        page_unlock(pp);
6011                                } else if (amp != NULL)
6012                                        anon_array_exit(&cookie);
6013                        }
6014                        VPP_SETPROT(svp, prot);
6015                        offset += PAGESIZE;
6016                }
6017                if (amp != NULL)
6018                        ANON_LOCK_EXIT(&amp->a_rwlock);

6020                /*
6021                 * Did we terminate prematurely?  If so, simply unload
6022                 * the translations to the things we've updated so far.
6023                 */
6024                if (svp != evp) {
6025                        if (unload_done) {
6026                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6027                                return (IE_NOMEM);
6028                        }
6029                        len = (svp - &svd->vpage[seg_page(seg, addr)]) *
6030                            PAGESIZE;
6031                        ASSERT(seg->s_szc == 0 || IS_P2ALIGNED(len, pgsz));
6032                        if (len != 0)
6033                                hat_unload(seg->s_as->a_hat, addr,
6034                                    len, HAT_UNLOAD);
6035                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6036                        return (IE_NOMEM);
```



```
6037                }
6038        } else {
6039                segvn_vpage(seg);
6040                if (svd->vpage == NULL) {
6041                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6042                        return (ENOMEM);
6043                }
6044                svd->pageprot = 1;
6045                evp = &svd->vpage[seg_page(seg, addr + len)];
6046                for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {
6047                        VPP_SETPROT(svp, prot);
6048                }
6049        }

6051        if (unload_done) {
6052                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6053                return (0);
6054        }

6056        if (((prot & PROT_WRITE) != 0 &&
6057            (svd->vp != NULL || svd->type == MAP_PRIVATE)) ||
6058            (prot & ~PROT_USER) == PROT_NONE) {
6059                /*
6060                 * Either private or shared data with write access (in
6061                 * which case we need to throw out all former translations
6062                 * so that we get the right translations set up on fault
6063                 * and we don't allow write access to any copy-on-write pages
6064                 * that might be around or to prevent write access to pages
6065                 * representing holes in a file), or we don't have permission
6066                 * to access the memory at all (in which case we have to
6067                 * unload any current translations that might exist).
6068                 */
6069                hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
6070        } else {
6071                /*
6072                 * A shared mapping or a private mapping in which write
6073                 * protection is going to be denied - just change all the
6074                 * protections over the range of addresses in question.
6075                 * segvn does not support any other attributes other
6076                 * than prot so we can use hat_chgattr.
6077                 */
6078                hat_chgattr(seg->s_as->a_hat, addr, len, prot);
6079        }

6081        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

6083        return (0);
6084 }

6086 /*
6087  * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
6088  * to determine if the seg is capable of mapping the requested szc.
6089  */
6090 static int
6091 segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
6092 {
6093        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6094        struct segvn_data *nsvd;
6095        struct anon_map *amp = svd->amp;
6096        struct seg *nseg;
6097        caddr_t eaddr = addr + len, a;
6098        size_t pgsz = page_get_pagesize(szc);
6099        pgcnt_t pgcnt = page_get_pagecnt(szc);
6100        int err;
6101        u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);
```

```
6103          ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
6103          ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6104          ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);

6106          if (seg->s_szc == szc || segvn_lpg_disable != 0) {
6107                return (0);
6108          }

6110          /*
6111           * addr should always be pgsz aligned but eaddr may be misaligned if
6112           * it's at the end of the segment.
6113           *
6114           * XXX we should assert this condition since as_setpagesize() logic
6115           * guarantees it.
6116           */
6117          if (!IS_P2ALIGNED(addr, pgsz) ||
6118              (!IS_P2ALIGNED(eaddr, pgsz) &&
6119              eaddr != seg->s_base + seg->s_size)) {

6121                segvn_setpgsz_align_err++;
6122                return (EINVAL);
6123          }

6125          if (amp != NULL && svd->type == MAP_SHARED) {
6126                ulong_t an_idx = svd->anon_index + seg_page(seg, addr);
6127                if (!IS_P2ALIGNED(an_idx, pgcnt)) {

6129                      segvn_setpgsz_anon_align_err++;
6130                      return (EINVAL);
6131                }
6132          }

6134          if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
6135              szc > segvn_maxpgszc) {
6136                return (EINVAL);
6137          }

6139          /* paranoid check */
6140          if (svd->vp != NULL &&
6141              (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
6142                return (EINVAL);
6143          }

6145          if (seg->s_szc == 0 && svd->vp != NULL &&
6146              map_addr_vacalign_check(addr, off)) {
6147                return (EINVAL);
6148          }

6150          /*
6151           * Check that protections are the same within new page
6152           * size boundaries.
6153           */
6154          if (svd->pageprot) {
6155                for (a = addr; a < eaddr; a += pgsz) {
6156                      if ((a + pgsz) > eaddr) {
6157                            if (!sameprot(seg, a, eaddr - a)) {
6158                                  return (EINVAL);
6159                            }
6160                      } else {
6161                            if (!sameprot(seg, a, pgsz)) {
6162                                  return (EINVAL);
6163                            }
6164                      }
6165                }
6166          }
```



```
6168          /*
6169           * Since we are changing page size we first have to flush
6170           * the cache. This makes sure all the pagelock calls have
6171           * to recheck protections.
6172           */
6173          if (svd->softlockcnt > 0) {
6174                ASSERT(svd->tr_state == SEGVN_TR_OFF);

6176                /*
6177                 * If this is shared segment non 0 softlockcnt
6178                 * means locked pages are still in use.
6179                 */
6180                if (svd->type == MAP_SHARED) {
6181                      return (EAGAIN);
6182                }

6184                /*
6185                 * Since we do have the segvn writers lock nobody can fill
6186                 * the cache with entries belonging to this seg during
6187                 * the purge. The flush either succeeds or we still have
6188                 * pending I/Os.
6189                 */
6190                segvn_purge(seg);
6191                if (svd->softlockcnt > 0) {
6192                      return (EAGAIN);
6193                }
6194          }

6196          if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6197                ASSERT(svd->amp == NULL);
6198                ASSERT(svd->tr_state == SEGVN_TR_OFF);
6199                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6200                    HAT_REGION_TEXT);
6201                svd->rcookie = HAT_INVALID_REGION_COOKIE;
6202          } else if (svd->tr_state == SEGVN_TR_INIT) {
6203                svd->tr_state = SEGVN_TR_OFF;
6204          } else if (svd->tr_state == SEGVN_TR_ON) {
6205                ASSERT(svd->amp != NULL);
6206                segvn_textunrepl(seg, 1);
6207                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6208                amp = NULL;
6209          }

6211          /*
6212           * Operation for sub range of existing segment.
6213           */
6214          if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6215                if (szc < seg->s_szc) {
6216                      VM_STAT_ADD(segvnvmstats.demoterange[2]);
6217                      err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6218                      if (err == 0) {
6219                            return (IE_RETRY);
6220                      }
6221                      if (err == ENOMEM) {
6222                            return (IE_NOMEM);
6223                      }
6224                      return (err);
6225                }
6226                if (addr != seg->s_base) {
6227                      nseg = segvn_split_seg(seg, addr);
6228                      if (eaddr != (nseg->s_base + nseg->s_size)) {
6229                            /* eaddr is szc aligned */
6230                            (void) segvn_split_seg(nseg, eaddr);
6231                      }
6232                      return (IE_RETRY);
6233                }
```

```
6234                if (eaddr != (seg->s_base + seg->s_size)) {
6235                        /* eaddr is szc aligned */
6236                        (void) segvn_split_seg(seg, eaddr);
6237                }
6238                return (IE_RETRY);
6239        }

6241        /*
6242         * Break any low level sharing and reset seg->s_szc to 0.
6243         */
6244        if ((err = segvn_clrszc(seg)) != 0) {
6245                if (err == ENOMEM) {
6246                        err = IE_NOMEM;
6247                }
6248                return (err);
6249        }
6250        ASSERT(seg->s_szc == 0);

6252        /*
6253         * If the end of the current segment is not pgsz aligned
6254         * then attempt to concatenate with the next segment.
6255         */
6256        if (!IS_P2ALIGNED(eaddr, pgsz)) {
6257                nseg = AS_SEGNEXT(seg->s_as, seg);
6258                if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6259                        return (ENOMEM);
6260                }
6261                if (nseg->s_ops != &segvn_ops) {
6262                        return (EINVAL);
6263                }
6264                nsvd = (struct segvn_data *)nseg->s_data;
6265                if (nsvd->softlockcnt > 0) {
6266                        /*
6267                         * If this is shared segment non 0 softlockcnt
6268                         * means locked pages are still in use.
6269                         */
6270                        if (nsvd->type == MAP_SHARED) {
6271                                return (EAGAIN);
6272                        }
6273                        segvn_purge(nseg);
6274                        if (nsvd->softlockcnt > 0) {
6275                                return (EAGAIN);
6276                        }
6277                }
6278                err = segvn_clrszc(nseg);
6279                if (err == ENOMEM) {
6280                        err = IE_NOMEM;
6281                }
6282                if (err != 0) {
6283                        return (err);
6284                }
6285                ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6286                err = segvn_concat(seg, nseg, 1);
6287                if (err == -1) {
6288                        return (EINVAL);
6289                }
6290                if (err == -2) {
6291                        return (IE_NOMEM);
6292                }
6293                return (IE_RETRY);
6294        }

6296        /*
6297         * May need to re-align anon array to
6298         * new szc.
6299         */
```



```
6300        if (amp != NULL) {
6301                if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6302                        struct anon_hdr *nahp;

6304                        ASSERT(svd->type == MAP_PRIVATE);

6306                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6307                        ASSERT(amp->refcnt == 1);
6308                        nahp = anon_create(btop(amp->size), ANON_NOSLEEP);
6309                        if (nahp == NULL) {
6310                                ANON_LOCK_EXIT(&amp->a_rwlock);
6311                                return (IE_NOMEM);
6312                        }
6313                        if (anon_copy_ptr(amp->ahp, svd->anon_index,
6314                            nahp, 0, btop(seg->s_size), ANON_NOSLEEP)) {
6315                                anon_release(nahp, btop(amp->size));
6316                                ANON_LOCK_EXIT(&amp->a_rwlock);
6317                                return (IE_NOMEM);
6318                        }
6319                        anon_release(amp->ahp, btop(amp->size));
6320                        amp->ahp = nahp;
6321                        svd->anon_index = 0;
6322                        ANON_LOCK_EXIT(&amp->a_rwlock);
6323                }
6324        }
6325        if (svd->vp != NULL && szc != 0) {
6326                struct vattr va;
6327                u_offset_t eoffpage = svd->offset;
6328                va.va_mask = AT_SIZE;
6329                eoffpage += seg->s_size;
6330                eoffpage = btopr(eoffpage);
6331                if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
6332                        segvn_setpgsz_getattr_err++;
6333                        return (EINVAL);
6334                }
6335                if (btopr(va.va_size) < eoffpage) {
6336                        segvn_setpgsz_eof_err++;
6337                        return (EINVAL);
6338                }
6339                if (amp != NULL) {
6340                        /*
6341                         * anon_fill_cow_holes() may call VOP_GETPAGE().
6342                         * don't take anon map lock here to avoid holding it
6343                         * across VOP_GETPAGE() calls that may call back into
6344                         * segvn for klsutering checks. We don't really need
6345                         * anon map lock here since it's a private segment and
6346                         * we hold as level lock as writers.
6347                         */
6348                        if ((err = anon_fill_cow_holes(seg, seg->s_base,
6349                            amp->ahp, svd->anon_index, svd->vp, svd->offset,
6350                            seg->s_size, szc, svd->prot, svd->vpage,
6351                            svd->cred)) != 0) {
6352                                return (EINVAL);
6353                        }
6354                }
6355                segvn_setvnode_mpss(svd->vp);
6356        }

6358        if (amp != NULL) {
6359                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6360                if (svd->type == MAP_PRIVATE) {
6361                        amp->a_szc = szc;
6362                } else if (szc > amp->a_szc) {
6363                        amp->a_szc = szc;
6364                }
6365                ANON_LOCK_EXIT(&amp->a_rwlock);
```

```
6366         }

6368         seg->s_szc = szc;

6370         return (0);
6371 }

6373 static int
6374 segvn_clrszc(struct seg *seg)
6375 {
6376         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6377         struct anon_map *amp = svd->amp;
6378         size_t pgsz;
6379         pgcnt_t pages;
6380         int err = 0;
6381         caddr_t a = seg->s_base;
6382         caddr_t ea = a + seg->s_size;
6383         ulong_t an_idx = svd->anon_index;
6384         vnode_t *vp = svd->vp;
6385         struct vpage *vpage = svd->vpage;
6386         page_t *anon_pl[1 + 1], *pp;
6387         struct anon *ap, *oldap;
6388         uint_t prot = svd->prot, vpprot;
6389         int pageflag = 0;

6391         ASSERT(AS_WRITE_HELD(seg->s_as) ||
6391         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
6392             SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
6393         ASSERT(svd->softlockcnt == 0);

6395         if (vp == NULL && amp == NULL) {
6396                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6397                 seg->s_szc = 0;
6398                 return (0);
6399         }

6401         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6402                 ASSERT(svd->amp == NULL);
6403                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
6404                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6405                     HAT_REGION_TEXT);
6406                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
6407         } else if (svd->tr_state == SEGVN_TR_ON) {
6408                 ASSERT(svd->amp != NULL);
6409                 segvn_textunrepl(seg, 1);
6410                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6411                 amp = NULL;
6412         } else {
6413                 if (svd->tr_state != SEGVN_TR_OFF) {
6414                         ASSERT(svd->tr_state == SEGVN_TR_INIT);
6415                         svd->tr_state = SEGVN_TR_OFF;
6416                 }

6418                 /*
6419                  * do HAT_UNLOAD_UNMAP since we are changing the pagesize.
6420                  * unload argument is 0 when we are freeing the segment
6421                  * and unload was already done.
6422                  */
6423                 hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
6424                     HAT_UNLOAD_UNMAP);
6425         }

6427         if (amp == NULL || svd->type == MAP_SHARED) {
6428                 seg->s_szc = 0;
6429                 return (0);
6430         }
```



```
6432         pgsz = page_get_pagesize(seg->s_szc);
6433         pages = btop(pgsz);

6435         /*
6436          * XXX anon rwlock is not really needed because this is a
6437          * private segment and we are writers.
6438          */
6439         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);

6441         for (; a < ea; a += pgsz, an_idx += pages) {
6442                 if ((oldap = anon_get_ptr(amp->ahp, an_idx)) != NULL) {
6443                         ASSERT(vpage != NULL || svd->pageprot == 0);
6444                         if (vpage != NULL) {
6445                                 ASSERT(sameprot(seg, a, pgsz));
6446                                 prot = VPP_PROT(vpage);
6447                                 pageflag = VPP_ISPPLOCK(vpage) ? LOCK_PAGE : 0;
6448                         }
6449                         if (seg->s_szc != 0) {
6450                                 ASSERT(vp == NULL || anon_pages(amp->ahp,
6451                                     an_idx, pages) == pages);
6452                                 if ((err = anon_map_demotepages(amp, an_idx,
6453                                     seg, a, prot, vpage, svd->cred)) != 0) {
6454                                         goto out;
6455                                 }
6456                         } else {
6457                                 if (oldap->an_refcnt == 1) {
6458                                         continue;
6459                                 }
6460                                 if ((err = anon_getpage(&oldap, &vpprot,
6461                                     anon_pl, PAGESIZE, seg, a, S_READ,
6462                                     svd->cred))) {
6463                                         goto out;
6464                                 }
6465                                 if ((pp = anon_private(&ap, seg, a, prot,
6466                                     anon_pl[0], pageflag, svd->cred)) == NULL) {
6467                                         err = ENOMEM;
6468                                         goto out;
6469                                 }
6470                                 anon_decref(oldap);
6471                                 (void) anon_set_ptr(amp->ahp, an_idx, ap,
6472                                     ANON_SLEEP);
6473                                 page_unlock(pp);
6474                         }
6475                 }
6476                 vpage = (vpage == NULL) ? NULL : vpage + pages;
6477         }

6479         amp->a_szc = 0;
6480         seg->s_szc = 0;
6481 out:
6482         ANON_LOCK_EXIT(&amp->a_rwlock);
6483         return (err);
6484 }
______unchanged_portion_omitted_

6590 /*
6591  * Returns right (upper address) segment if split occurred.
6592  * If the address is equal to the beginning or end of its segment it returns
6593  * the current segment.
6594  */
6595 static struct seg *
6596 segvn_split_seg(struct seg *seg, caddr_t addr)
6597 {
6598         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6599         struct seg *nseg;
```

```
6600         size_t nsize;
6601         struct segvn_data *nsvd;

6603         ASSERT(AS_WRITE_HELD(seg->s_as));
6603         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6604         ASSERT(svd->tr_state == SEGVN_TR_OFF);

6606         ASSERT(addr >= seg->s_base);
6607         ASSERT(addr <= seg->s_base + seg->s_size);
6608         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);

6610         if (addr == seg->s_base || addr == seg->s_base + seg->s_size)
6611                 return (seg);

6613         nsize = seg->s_base + seg->s_size - addr;
6614         seg->s_size = addr - seg->s_base;
6615         nseg = seg_alloc(seg->s_as, addr, nsize);
6616         ASSERT(nseg != NULL);
6617         nseg->s_ops = seg->s_ops;
6618         nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
6619         nseg->s_data = (void *)nsvd;
6620         nseg->s_szc = seg->s_szc;
6621         *nsvd = *svd;
6622         ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6623         nsvd->seg = nseg;
6624         rw_init(&nsvd->lock, NULL, RW_DEFAULT, NULL);

6626         if (nsvd->vp != NULL) {
6627                 VN_HOLD(nsvd->vp);
6628                 nsvd->offset = svd->offset +
6629                     (uintptr_t)(nseg->s_base - seg->s_base);
6630                 if (nsvd->type == MAP_SHARED)
6631                         lgrp_shm_policy_init(NULL, nsvd->vp);
6632         } else {
6633                 /*
6634                  * The offset for an anonymous segment has no signifigance in
6635                  * terms of an offset into a file. If we were to use the above
6636                  * calculation instead, the structures read out of
6637                  * /proc/<pid>/xmap would be more difficult to decipher since
6638                  * it would be unclear whether two seemingly contiguous
6639                  * prxmap_t structures represented different segments or a
6640                  * single segment that had been split up into multiple prxmap_t
6641                  * structures (e.g. if some part of the segment had not yet
6642                  * been faulted in).
6643                  */
6644                 nsvd->offset = 0;
6645         }

6647         ASSERT(svd->softlockcnt == 0);
6648         ASSERT(svd->softlockcnt_sbase == 0);
6649         ASSERT(svd->softlockcnt_send == 0);
6650         crhold(svd->cred);

6652         if (svd->vpage != NULL) {
6653                 size_t bytes = vpgtob(seg_pages(seg));
6654                 size_t nbytes = vpgtob(seg_pages(nseg));
6655                 struct vpage *ovpage = svd->vpage;

6657                 svd->vpage = kmem_alloc(bytes, KM_SLEEP);
6658                 bcopy(ovpage, svd->vpage, bytes);
6659                 nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);
6660                 bcopy(ovpage + seg_pages(seg), nsvd->vpage, nbytes);
6661                 kmem_free(ovpage, bytes + nbytes);
6662         }
6663         if (svd->amp != NULL && svd->type == MAP_PRIVATE) {
6664                 struct anon_map *oamp = svd->amp, *namp;
```



```
6665                 struct anon_hdr *nahp;

6667                 ANON_LOCK_ENTER(&oamp->a_rwlock, RW_WRITER);
6668                 ASSERT(oamp->refcnt == 1);
6669                 nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
6670                 (void) anon_copy_ptr(oamp->ahp, svd->anon_index,
6671                     nahp, 0, btop(seg->s_size), ANON_SLEEP);

6673                 namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
6674                 namp->a_szc = nseg->s_szc;
6675                 (void) anon_copy_ptr(oamp->ahp,
6676                     svd->anon_index + btop(seg->s_size),
6677                     namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
6678                 anon_release(oamp->ahp, btop(oamp->size));
6679                 oamp->ahp = nahp;
6680                 oamp->size = seg->s_size;
6681                 svd->anon_index = 0;
6682                 nsvd->amp = namp;
6683                 nsvd->anon_index = 0;
6684                 ANON_LOCK_EXIT(&oamp->a_rwlock);
6685         } else if (svd->amp != NULL) {
6686                 pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
6687                 ASSERT(svd->amp == nsvd->amp);
6688                 ASSERT(seg->s_szc <= svd->amp->a_szc);
6689                 nsvd->anon_index = svd->anon_index + seg_pages(seg);
6690                 ASSERT(IS_P2ALIGNED(nsvd->anon_index, pgcnt));
6691                 ANON_LOCK_ENTER(&svd->amp->a_rwlock, RW_WRITER);
6692                 svd->amp->refcnt++;
6693                 ANON_LOCK_EXIT(&svd->amp->a_rwlock);
6694         }

6696         /*
6697          * Split the amount of swap reserved.
6698          */
6699         if (svd->swresv) {
6700                 /*
6701                  * For MAP_NORESERVE, only allocate swap reserve for pages
6702                  * being used.  Other segments get enough to cover whole
6703                  * segment.
6704                  */
6705                 if (svd->flags & MAP_NORESERVE) {
6706                         size_t  oswresv;

6708                         ASSERT(svd->amp);
6709                         oswresv = svd->swresv;
6710                         svd->swresv = ptob(anon_pages(svd->amp->ahp,
6711                             svd->anon_index, btop(seg->s_size)));
6712                         nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
6713                             nsvd->anon_index, btop(nseg->s_size)));
6714                         ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
6715                 } else {
6716                         if (svd->pageswap) {
6717                                 svd->swresv = segvn_count_swap_by_vpages(seg);
6718                                 ASSERT(nsvd->swresv >= svd->swresv);
6719                                 nsvd->swresv -= svd->swresv;
6720                         } else {
6721                                 ASSERT(svd->swresv == seg->s_size +
6722                                     nseg->s_size);
6723                                 svd->swresv = seg->s_size;
6724                                 nsvd->swresv = nseg->s_size;
6725                         }
6726                 }
6727         }

6729         return (nseg);
6730 }
```

```
6732 /*
6733  * called on memory operations (unmap, setprot, setpagesize) for a subset
6734  * of a large page segment to either demote the memory range (SDR_RANGE)
6735  * or the ends (SDR_END) by addr/len.
6736  *
6737  * returns 0 on success. returns errno, including ENOMEM, on failure.
6738  */
6739 static int
6740 segvn_demote_range(
6741         struct seg *seg,
6742         caddr_t addr,
6743         size_t len,
6744         int flag,
6745         uint_t szcvec)
6746 {
6747         caddr_t eaddr = addr + len;
6748         caddr_t lpgaddr, lpgeaddr;
6749         struct seg *nseg;
6750         struct seg *badseg1 = NULL;
6751         struct seg *badseg2 = NULL;
6752         size_t pgsz;
6753         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6754         int err;
6755         uint_t szc = seg->s_szc;
6756         uint_t tszcvec;

6758         ASSERT(AS_WRITE_HELD(seg->s_as));
6758         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6759         ASSERT(svd->tr_state == SEGVN_TR_OFF);
6760         ASSERT(szc != 0);
6761         pgsz = page_get_pagesize(szc);
6762         ASSERT(seg->s_base != addr || seg->s_size != len);
6763         ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);
6764         ASSERT(svd->softlockcnt == 0);
6765         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6766         ASSERT(szcvec == 0 || (flag == SDR_END && svd->type == MAP_SHARED));

6768         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
6769         ASSERT(flag == SDR_RANGE || eaddr < lpgeaddr || addr > lpgaddr);
6770         if (flag == SDR_RANGE) {
6771                 /* demote entire range */
6772                 badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
6773                 (void) segvn_split_seg(nseg, lpgeaddr);
6774                 ASSERT(badseg1->s_base == lpgaddr);
6775                 ASSERT(badseg1->s_size == lpgeaddr - lpgaddr);
6776         } else if (addr != lpgaddr) {
6777                 ASSERT(flag == SDR_END);
6778                 badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
6779                 if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz &&
6780                     eaddr < lpgaddr + 2 * pgsz) {
6781                         (void) segvn_split_seg(nseg, lpgeaddr);
6782                         ASSERT(badseg1->s_base == lpgaddr);
6783                         ASSERT(badseg1->s_size == 2 * pgsz);
6784                 } else {
6785                         nseg = segvn_split_seg(nseg, lpgaddr + pgsz);
6786                         ASSERT(badseg1->s_base == lpgaddr);
6787                         ASSERT(badseg1->s_size == pgsz);
6788                         if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz) {
6789                                 ASSERT(lpgeaddr - lpgaddr > 2 * pgsz);
6790                                 nseg = segvn_split_seg(nseg, lpgeaddr - pgsz);
6791                                 badseg2 = nseg;
6792                                 (void) segvn_split_seg(nseg, lpgeaddr);
6793                                 ASSERT(badseg2->s_base == lpgeaddr - pgsz);
6794                                 ASSERT(badseg2->s_size == pgsz);
6795                         }
```



```
6796                 }
6797         } else {
6798                 ASSERT(flag == SDR_END);
6799                 ASSERT(eaddr < lpgeaddr);
6800                 badseg1 = nseg = segvn_split_seg(seg, lpgeaddr - pgsz);
6801                 (void) segvn_split_seg(nseg, lpgeaddr);
6802                 ASSERT(badseg1->s_base == lpgeaddr - pgsz);
6803                 ASSERT(badseg1->s_size == pgsz);
6804         }

6806         ASSERT(badseg1 != NULL);
6807         ASSERT(badseg1->s_szc == szc);
6808         ASSERT(flag == SDR_RANGE || badseg1->s_size == pgsz ||
6809             badseg1->s_size == 2 * pgsz);
6810         ASSERT(sameprot(badseg1, badseg1->s_base, pgsz));
6811         ASSERT(badseg1->s_size == pgsz ||
6812             sameprot(badseg1, badseg1->s_base + pgsz, pgsz));
6813         if (err = segvn_clrszc(badseg1)) {
6814                 return (err);
6815         }
6816         ASSERT(badseg1->s_szc == 0);

6818         if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
6819                 uint_t tszc = highbit(tszcvec) - 1;
6820                 caddr_t ta = MAX(addr, badseg1->s_base);
6821                 caddr_t te;
6822                 size_t tpgsz = page_get_pagesize(tszc);

6824                 ASSERT(svd->type == MAP_SHARED);
6825                 ASSERT(flag == SDR_END);
6826                 ASSERT(tszc < szc && tszc > 0);

6828                 if (eaddr > badseg1->s_base + badseg1->s_size) {
6829                         te = badseg1->s_base + badseg1->s_size;
6830                 } else {
6831                         te = eaddr;
6832                 }

6834                 ASSERT(ta <= te);
6835                 badseg1->s_szc = tszc;
6836                 if (!IS_P2ALIGNED(ta, tpgsz) || !IS_P2ALIGNED(te, tpgsz)) {
6837                         if (badseg2 != NULL) {
6838                                 err = segvn_demote_range(badseg1, ta, te - ta,
6839                                     SDR_END, tszcvec);
6840                                 if (err != 0) {
6841                                         return (err);
6842                                 }
6843                         } else {
6844                                 return (segvn_demote_range(badseg1, ta,
6845                                     te - ta, SDR_END, tszcvec));
6846                         }
6847                 }
6848         }

6850         if (badseg2 == NULL)
6851                 return (0);
6852         ASSERT(badseg2->s_szc == szc);
6853         ASSERT(badseg2->s_size == pgsz);
6854         ASSERT(sameprot(badseg2, badseg2->s_base, badseg2->s_size));
6855         if (err = segvn_clrszc(badseg2)) {
6856                 return (err);
6857         }
6858         ASSERT(badseg2->s_szc == 0);

6860         if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
6861                 uint_t tszc = highbit(tszcvec) - 1;
```

```
6862                 size_t tpgsz = page_get_pagesize(tszc);

6864                 ASSERT(svd->type == MAP_SHARED);
6865                 ASSERT(flag == SDR_END);
6866                 ASSERT(tszc < szc && tszc > 0);
6867                 ASSERT(badseg2->s_base > addr);
6868                 ASSERT(eaddr > badseg2->s_base);
6869                 ASSERT(eaddr < badseg2->s_base + badseg2->s_size);

6871                 badseg2->s_szc = tszc;
6872                 if (!IS_P2ALIGNED(eaddr, tpgsz)) {
6873                         return (segvn_demote_range(badseg2, badseg2->s_base,
6874                             eaddr - badseg2->s_base, SDR_END, tszcvec));
6875                 }
6876         }

6878         return (0);
6879 }

6881 static int
6882 segvn_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
6883 {
6884         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6885         struct vpage *vp, *evp;

6887         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6887         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6889         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
6890         /*
6891          * If segment protection can be used, simply check against them.
6892          */
6893         if (svd->pageprot == 0) {
6894                 int err;

6896                 err = ((svd->prot & prot) != prot) ? EACCES : 0;
6897                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6898                 return (err);
6899         }

6901         /*
6902          * Have to check down to the vpage level.
6903          */
6904         evp = &svd->vpage[seg_page(seg, addr + len)];
6905         for (vp = &svd->vpage[seg_page(seg, addr)]; vp < evp; vp++) {
6906                 if ((VPP_PROT(vp) & prot) != prot) {
6907                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6908                         return (EACCES);
6909                 }
6910         }
6911         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6912         return (0);
6913 }

6915 static int
6916 segvn_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
6917 {
6918         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6919         size_t pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;

6921         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6921         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6923         if (pgno != 0) {
6924                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
6925                 if (svd->pageprot == 0) {
```



```
6926                         do {
6927                                 protv[--pgno] = svd->prot;
6928                         } while (pgno != 0);
6929                 } else {
6930                         size_t pgoff = seg_page(seg, addr);

6932                         do {
6933                                 pgno--;
6934                                 protv[pgno] = VPP_PROT(&svd->vpage[pgno+pgoff]);
6935                         } while (pgno != 0);
6936                 }
6937                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6938         }
6939         return (0);
6940 }

6942 static u_offset_t
6943 segvn_getoffset(struct seg *seg, caddr_t addr)
6944 {
6945         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6947         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6947         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6949         return (svd->offset + (uintptr_t)(addr - seg->s_base));
6950 }

6952 /*ARGSUSED*/
6953 static int
6954 segvn_gettype(struct seg *seg, caddr_t addr)
6955 {
6956         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6958         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6958         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6960         return (svd->type | (svd->flags & (MAP_NORESERVE | MAP_TEXT |
6961             MAP_INITDATA)));
6962 }

6964 /*ARGSUSED*/
6965 static int
6966 segvn_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
6967 {
6968         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6970         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6970         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6972         *vpp = svd->vp;
6973         return (0);
6974 }

6976 /*
6977  * Check to see if it makes sense to do kluster/read ahead to
6978  * addr + delta relative to the mapping at addr.  We assume here
6979  * that delta is a signed PAGESIZE'd multiple (which can be negative).
6980  *
6981  * For segvn, we currently "approve" of the action if we are
6982  * still in the segment and it maps from the same vp/off,
6983  * or if the advice stored in segvn_data or vpages allows it.
6984  * Currently, klustering is not allowed only if MADV_RANDOM is set.
6985  */
6986 static int
6987 segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
6988 {
```

```
6989         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6990         struct anon *oap, *ap;
6991         ssize_t pd;
6992         size_t page;
6993         struct vnode *vp1, *vp2;
6994         u_offset_t off1, off2;
6995         struct anon_map *amp;

6997         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6998         ASSERT(AS_WRITE_HELD(seg->s_as) ||
6997         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6998         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
6999             SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

7001         if (addr + delta < seg->s_base ||
7002             addr + delta >= (seg->s_base + seg->s_size))
7003                 return (-1);            /* exceeded segment bounds */

7005         pd = delta / (ssize_t)PAGESIZE; /* divide to preserve sign bit */
7006         page = seg_page(seg, addr);

7008         /*
7009          * Check to see if either of the pages addr or addr + delta
7010          * have advice set that prevents klustering (if MADV_RANDOM advice
7011          * is set for entire segment, or MADV_SEQUENTIAL is set and delta
7012          * is negative).
7013          */
7014         if (svd->advice == MADV_RANDOM ||
7015             svd->advice == MADV_SEQUENTIAL && delta < 0)
7016                 return (-1);
7017         else if (svd->pageadvice && svd->vpage) {
7018                 struct vpage *bvpp, *evpp;

7020                 bvpp = &svd->vpage[page];
7021                 evpp = &svd->vpage[page + pd];
7022                 if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
7023                     VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
7024                         return (-1);
7025                 if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
7026                     VPP_ADVICE(evpp) == MADV_RANDOM)
7027                         return (-1);
7028         }

7030         if (svd->type == MAP_SHARED)
7031                 return (0);             /* shared mapping - all ok */

7033         if ((amp = svd->amp) == NULL)
7034                 return (0);             /* off original vnode */

7036         page += svd->anon_index;

7038         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

7040         oap = anon_get_ptr(amp->ahp, page);
7041         ap = anon_get_ptr(amp->ahp, page + pd);

7043         ANON_LOCK_EXIT(&amp->a_rwlock);

7045         if ((oap == NULL && ap != NULL) || (oap != NULL && ap == NULL)) {
7046                 return (-1);            /* one with and one without an anon */
7047         }

7049         if (oap == NULL) {              /* implies that ap == NULL */
7050                 return (0);             /* off original vnode */
7051         }
```



```
7053         /*
7054          * Now we know we have two anon pointers - check to
7055          * see if they happen to be properly allocated.
7056          */

7058         /*
7059          * XXX We cheat here and don't lock the anon slots. We can't because
7060          * we may have been called from the anon layer which might already
7061          * have locked them. We are holding a refcnt on the slots so they
7062          * can't disappear. The worst that will happen is we'll get the wrong
7063          * names (vp, off) for the slots and make a poor klustering decision.
7064          */
7065         swap_xlate(ap, &vp1, &off1);
7066         swap_xlate(oap, &vp2, &off2);


7069         if (!VOP_CMP(vp1, vp2, NULL) || off1 - off2 != delta)
7070                 return (-1);
7071         return (0);
7072 }

7074 /*
7075  * Swap the pages of seg out to secondary storage, returning the
7076  * number of bytes of storage freed.
7077  *
7078  * The basic idea is first to unload all translations and then to call
7079  * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
7080  * swap device.  Pages to which other segments have mappings will remain
7081  * mapped and won't be swapped.  Our caller (as_swapout) has already
7082  * performed the unloading step.
7083  *
7084  * The value returned is intended to correlate well with the process's
7085  * memory requirements.  However, there are some caveats:
7086  * 1)   When given a shared segment as argument, this routine will
7087  *      only succeed in swapping out pages for the last sharer of the
7088  *      segment.  (Previous callers will only have decremented mapping
7089  *      reference counts.)
7090  * 2)   We assume that the hat layer maintains a large enough translation
7091  *      cache to capture process reference patterns.
7092  */
7093 static size_t
7094 segvn_swapout(struct seg *seg)
7095 {
7096         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7097         struct anon_map *amp;
7098         pgcnt_t pgcnt = 0;
7099         pgcnt_t npages;
7100         pgcnt_t page;
7101         ulong_t anon_index;

7103         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
7103         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7105         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
7106         /*
7107          * Find pages unmapped by our caller and force them
7108          * out to the virtual swap device.
7109          */
7110         if ((amp = svd->amp) != NULL)
7111                 anon_index = svd->anon_index;
7112         npages = seg->s_size >> PAGESHIFT;
7113         for (page = 0; page < npages; page++) {
7114                 page_t *pp;
7115                 struct anon *ap;
7116                 struct vnode *vp;
7117                 u_offset_t off;
```

```
7118                anon_sync_obj_t cookie;

7120                /*
7121                 * Obtain <vp, off> pair for the page, then look it up.
7122                 *
7123                 * Note that this code is willing to consider regular
7124                 * pages as well as anon pages.  Is this appropriate here?
7125                 */
7126                ap = NULL;
7127                if (amp != NULL) {
7128                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7129                        if (anon_array_try_enter(amp, anon_index + page,
7130                            &cookie)) {
7131                                ANON_LOCK_EXIT(&amp->a_rwlock);
7132                                continue;
7133                        }
7134                        ap = anon_get_ptr(amp->ahp, anon_index + page);
7135                        if (ap != NULL) {
7136                                swap_xlate(ap, &vp, &off);
7137                        } else {
7138                                vp = svd->vp;
7139                                off = svd->offset + ptob(page);
7140                        }
7141                        anon_array_exit(&cookie);
7142                        ANON_LOCK_EXIT(&amp->a_rwlock);
7143                } else {
7144                        vp = svd->vp;
7145                        off = svd->offset + ptob(page);
7146                }
7147                if (vp == NULL) {               /* untouched zfod page */
7148                        ASSERT(ap == NULL);
7149                        continue;
7150                }

7152                pp = page_lookup_nowait(vp, off, SE_SHARED);
7153                if (pp == NULL)
7154                        continue;


7157                /*
7158                 * Examine the page to see whether it can be tossed out,
7159                 * keeping track of how many we've found.
7160                 */
7161                if (!page_tryupgrade(pp)) {
7162                        /*
7163                         * If the page has an i/o lock and no mappings,
7164                         * it's very likely that the page is being
7165                         * written out as a result of klustering.
7166                         * Assume this is so and take credit for it here.
7167                         */
7168                        if (!page_io_trylock(pp)) {
7169                                if (!hat_page_is_mapped(pp))
7170                                        pgcnt++;
7171                        } else {
7172                                page_io_unlock(pp);
7173                        }
7174                        page_unlock(pp);
7175                        continue;
7176                }
7177                ASSERT(!page_iolock_assert(pp));


7180                /*
7181                 * Skip if page is locked or has mappings.
7182                 * We don't need the page_struct_lock to look at lckcnt
7183                 * and cowcnt because the page is exclusive locked.
```



```
7184                 */
7185                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0 ||
7186                    hat_page_is_mapped(pp)) {
7187                        page_unlock(pp);
7188                        continue;
7189                }

7191                /*
7192                 * dispose skips large pages so try to demote first.
7193                 */
7194                if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
7195                        page_unlock(pp);
7196                        /*
7197                         * XXX should skip the remaining page_t's of this
7198                         * large page.
7199                         */
7200                        continue;
7201                }

7203                ASSERT(pp->p_szc == 0);

7205                /*
7206                 * No longer mapped -- we can toss it out.  How
7207                 * we do so depends on whether or not it's dirty.
7208                 */
7209                if (hat_ismod(pp) && pp->p_vnode) {
7210                        /*
7211                         * We must clean the page before it can be
7212                         * freed.  Setting B_FREE will cause pvn_done
7213                         * to free the page when the i/o completes.
7214                         * XXX: This also causes it to be accounted
7215                         *      as a pageout instead of a swap: need
7216                         *      B_SWAPOUT bit to use instead of B_FREE.
7217                         *
7218                         * Hold the vnode before releasing the page lock
7219                         * to prevent it from being freed and re-used by
7220                         * some other thread.
7221                         */
7222                        VN_HOLD(vp);
7223                        page_unlock(pp);

7225                        /*
7226                         * Queue all i/o requests for the pageout thread
7227                         * to avoid saturating the pageout devices.
7228                         */
7229                        if (!queue_io_request(vp, off))
7230                                VN_RELE(vp);
7231                } else {
7232                        /*
7233                         * The page was clean, free it.
7234                         *
7235                         * XXX: Can we ever encounter modified pages
7236                         *      with no associated vnode here?
7237                         */
7238                        ASSERT(pp->p_vnode != NULL);
7239                        /*LINTED: constant in conditional context*/
7240                        VN_DISPOSE(pp, B_FREE, 0, kcred);
7241                }

7243                /*
7244                 * Credit now even if i/o is in progress.
7245                 */
7246                pgcnt++;
7247        }
7248        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
```

```
7250         /*
7251          * Wakeup pageout to initiate i/o on all queued requests.
7252          */
7253         cv_signal_pageout();
7254         return (ptob(pgcnt));
7255 }

7257 /*
7258  * Synchronize primary storage cache with real object in virtual memory.
7259  *
7260  * XXX - Anonymous pages should not be sync'ed out at all.
7261  */
7262 static int
7263 segvn_sync(struct seg *seg, caddr_t addr, size_t len, int attr, uint_t flags)
7264 {
7265         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7266         struct vpage *vpp;
7267         page_t *pp;
7268         u_offset_t offset;
7269         struct vnode *vp;
7270         u_offset_t off;
7271         caddr_t eaddr;
7272         int bflags;
7273         int err = 0;
7274         int segtype;
7275         int pageprot;
7276         int prot;
7277         ulong_t anon_index;
7278         struct anon_map *amp;
7279         struct anon *ap;
7280         anon_sync_obj_t cookie;

7282         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
7282         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7284         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

7286         if (svd->softlockcnt > 0) {
7287                 /*
7288                  * If this is shared segment non 0 softlockcnt
7289                  * means locked pages are still in use.
7290                  */
7291                 if (svd->type == MAP_SHARED) {
7292                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7293                         return (EAGAIN);
7294                 }

7296                 /*
7297                  * flush all pages from seg cache
7298                  * otherwise we may deadlock in swap_putpage
7299                  * for B_INVAL page (4175402).
7300                  *
7301                  * Even if we grab segvn WRITER's lock
7302                  * here, there might be another thread which could've
7303                  * successfully performed lookup/insert just before
7304                  * we acquired the lock here.  So, grabbing either
7305                  * lock here is of not much use.  Until we devise
7306                  * a strategy at upper layers to solve the
7307                  * synchronization issues completely, we expect
7308                  * applications to handle this appropriately.
7309                  */
7310                 segvn_purge(seg);
7311                 if (svd->softlockcnt > 0) {
7312                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7313                         return (EAGAIN);
7314                 }
```



```
7315         } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
7316             svd->amp->a_softlockcnt > 0) {
7317                 /*
7318                  * Try to purge this amp's entries from pcache. It will
7319                  * succeed only if other segments that share the amp have no
7320                  * outstanding softlock's.
7321                  */
7322                 segvn_purge(seg);
7323                 if (svd->amp->a_softlockcnt > 0 || svd->softlockcnt > 0) {
7324                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7325                         return (EAGAIN);
7326                 }
7327         }

7329         vpp = svd->vpage;
7330         offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7331         bflags = ((flags & MS_ASYNC) ? B_ASYNC : 0) |
7332             ((flags & MS_INVALIDATE) ? B_INVAL : 0);

7334         if (attr) {
7335                 pageprot = attr & ~(SHARED|PRIVATE);
7336                 segtype = (attr & SHARED) ? MAP_SHARED : MAP_PRIVATE;

7338                 /*
7339                  * We are done if the segment types don't match
7340                  * or if we have segment level protections and
7341                  * they don't match.
7342                  */
7343                 if (svd->type != segtype) {
7344                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7345                         return (0);
7346                 }
7347                 if (vpp == NULL) {
7348                         if (svd->prot != pageprot) {
7349                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7350                                 return (0);
7351                         }
7352                         prot = svd->prot;
7353                 } else
7354                         vpp = &svd->vpage[seg_page(seg, addr)];

7356         } else if (svd->vp && svd->amp == NULL &&
7357             (flags & MS_INVALIDATE) == 0) {

7359                 /*
7360                  * No attributes, no anonymous pages and MS_INVALIDATE flag
7361                  * is not on, just use one big request.
7362                  */
7363                 err = VOP_PUTPAGE(svd->vp, (offset_t)offset, len,
7364                     bflags, svd->cred, NULL);
7365                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7366                 return (err);
7367         }

7369         if ((amp = svd->amp) != NULL)
7370                 anon_index = svd->anon_index + seg_page(seg, addr);

7372         for (eaddr = addr + len; addr < eaddr; addr += PAGESIZE) {
7373                 ap = NULL;
7374                 if (amp != NULL) {
7375                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7376                         anon_array_enter(amp, anon_index, &cookie);
7377                         ap = anon_get_ptr(amp->ahp, anon_index++);
7378                         if (ap != NULL) {
7379                                 swap_xlate(ap, &vp, &off);
7380                         } else {
```

```
7381                                    vp = svd->vp;
7382                                    off = offset;
7383                            }
7384                            anon_array_exit(&cookie);
7385                            ANON_LOCK_EXIT(&amp->a_rwlock);
7386                    } else {
7387                            vp = svd->vp;
7388                            off = offset;
7389                    }
7390                    offset += PAGESIZE;

7392                    if (vp == NULL)         /* untouched zfod page */
7393                            continue;

7395                    if (attr) {
7396                            if (vpp) {
7397                                    prot = VPP_PROT(vpp);
7398                                    vpp++;
7399                            }
7400                            if (prot != pageprot) {
7401                                    continue;
7402                            }
7403                    }

7405                    /*
7406                     * See if any of these pages are locked --  if so, then we
7407                     * will have to truncate an invalidate request at the first
7408                     * locked one. We don't need the page_struct_lock to test
7409                     * as this is only advisory; even if we acquire it someone
7410                     * might race in and lock the page after we unlock and before
7411                     * we do the PUTPAGE, then PUTPAGE simply does nothing.
7412                     */
7413                    if (flags & MS_INVALIDATE) {
7414                            if ((pp = page_lookup(vp, off, SE_SHARED)) != NULL) {
7415                                    if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0) {
7416                                            page_unlock(pp);
7417                                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7418                                            return (EBUSY);
7419                                    }
7420                                    if (ap != NULL && pp->p_szc != 0 &&
7421                                        page_tryupgrade(pp)) {
7422                                            if (pp->p_lckcnt == 0 &&
7423                                                pp->p_cowcnt == 0) {
7424                                                    /*
7425                                                     * swapfs VN_DISPOSE() won't
7426                                                     * invalidate large pages.
7427                                                     * Attempt to demote.
7428                                                     * XXX can't help it if it
7429                                                     * fails. But for swapfs
7430                                                     * pages it is no big deal.
7431                                                     */
7432                                                    (void) page_try_demote_pages(
7433                                                        pp);
7434                                            }
7435                                    }
7436                                    page_unlock(pp);
7437                            }
7438                    } else if (svd->type == MAP_SHARED && amp != NULL) {
7439                            /*
7440                             * Avoid writing out to disk ISM's large pages
7441                             * because segspt_free_pages() relies on NULL an_pvp
7442                             * of anon slots of such pages.
7443                             */

7445                            ASSERT(svd->vp == NULL);
7446                            /*
```



```
7447                             * swapfs uses page_lookup_nowait if not freeing or
7448                             * invalidating and skips a page if
7449                             * page_lookup_nowait returns NULL.
7450                             */
7451                            pp = page_lookup_nowait(vp, off, SE_SHARED);
7452                            if (pp == NULL) {
7453                                    continue;
7454                            }
7455                            if (pp->p_szc != 0) {
7456                                    page_unlock(pp);
7457                                    continue;
7458                            }

7460                            /*
7461                             * Note ISM pages are created large so (vp, off)'s
7462                             * page cannot suddenly become large after we unlock
7463                             * pp.
7464                             */
7465                            page_unlock(pp);
7466                    }
7467                    /*
7468                     * XXX - Should ultimately try to kluster
7469                     * calls to VOP_PUTPAGE() for performance.
7470                     */
7471                    VN_HOLD(vp);
7472                    err = VOP_PUTPAGE(vp, (offset_t)off, PAGESIZE,
7473                        (bflags | (IS_SWAPFSVP(vp) ? B_PAGE_NOWAIT : 0)),
7474                        svd->cred, NULL);

7476                    VN_RELE(vp);
7477                    if (err)
7478                            break;
7479            }
7480            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7481            return (err);
7482    }

7484    /*
7485     * Determine if we have data corresponding to pages in the
7486     * primary storage virtual memory cache (i.e., "in core").
7487     */
7488    static size_t
7489    segvn_incore(struct seg *seg, caddr_t addr, size_t len, char *vec)
7490    {
7491            struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7492            struct vnode *vp, *avp;
7493            u_offset_t offset, aoffset;
7494            size_t p, ep;
7495            int ret;
7496            struct vpage *vpp;
7497            page_t *pp;
7498            uint_t start;
7499            struct anon_map *amp;           /* XXX - for locknest */
7500            struct anon *ap;
7501            uint_t attr;
7502            anon_sync_obj_t cookie;

7504            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
7504            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7506            SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
7507            if (svd->amp == NULL && svd->vp == NULL) {
7508                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7509                    bzero(vec, btopr(len));
7510                    return (len);   /* no anonymous pages created yet */
7511            }
```

```
7513         p = seg_page(seg, addr);
7514         ep = seg_page(seg, addr + len);
7515         start = svd->vp ? SEG_PAGE_VNODEBACKED : 0;

7517         amp = svd->amp;
7518         for (; p < ep; p++, addr += PAGESIZE) {
7519                 vpp = (svd->vpage) ? &svd->vpage[p]: NULL;
7520                 ret = start;
7521                 ap = NULL;
7522                 avp = NULL;
7523                 /* Grab the vnode/offset for the anon slot */
7524                 if (amp != NULL) {
7525                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7526                         anon_array_enter(amp, svd->anon_index + p, &cookie);
7527                         ap = anon_get_ptr(amp->ahp, svd->anon_index + p);
7528                         if (ap != NULL) {
7529                                 swap_xlate(ap, &avp, &aoffset);
7530                         }
7531                         anon_array_exit(&cookie);
7532                         ANON_LOCK_EXIT(&amp->a_rwlock);
7533                 }
7534                 if ((avp != NULL) && page_exists(avp, aoffset)) {
7535                         /* A page exists for the anon slot */
7536                         ret |= SEG_PAGE_INCORE;

7538                         /*
7539                          * If page is mapped and writable
7540                          */
7541                         attr = (uint_t)0;
7542                         if ((hat_getattr(seg->s_as->a_hat, addr,
7543                             &attr) != -1) && (attr & PROT_WRITE)) {
7544                                 ret |= SEG_PAGE_ANON;
7545                         }
7546                         /*
7547                          * Don't get page_struct lock for lckcnt and cowcnt,
7548                          * since this is purely advisory.
7549                          */
7550                         if ((pp = page_lookup_nowait(avp, aoffset,
7551                             SE_SHARED)) != NULL) {
7552                                 if (pp->p_lckcnt)
7553                                         ret |= SEG_PAGE_SOFTLOCK;
7554                                 if (pp->p_cowcnt)
7555                                         ret |= SEG_PAGE_HASCOW;
7556                                 page_unlock(pp);
7557                         }
7558                 }

7560                 /* Gather vnode statistics */
7561                 vp = svd->vp;
7562                 offset = svd->offset + (uintptr_t)(addr - seg->s_base);

7564                 if (vp != NULL) {
7565                         /*
7566                          * Try to obtain a "shared" lock on the page
7567                          * without blocking.  If this fails, determine
7568                          * if the page is in memory.
7569                          */
7570                         pp = page_lookup_nowait(vp, offset, SE_SHARED);
7571                         if ((pp == NULL) && (page_exists(vp, offset))) {
7572                                 /* Page is incore, and is named */
7573                                 ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
7574                         }
7575                         /*
7576                          * Don't get page_struct lock for lckcnt and cowcnt,
7577                          * since this is purely advisory.
```



```
7578                          */
7579                         if (pp != NULL) {
7580                                 ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
7581                                 if (pp->p_lckcnt)
7582                                         ret |= SEG_PAGE_SOFTLOCK;
7583                                 if (pp->p_cowcnt)
7584                                         ret |= SEG_PAGE_HASCOW;
7585                                 page_unlock(pp);
7586                         }
7587                 }

7589                 /* Gather virtual page information */
7590                 if (vpp) {
7591                         if (VPP_ISPPLOCK(vpp))
7592                                 ret |= SEG_PAGE_LOCKED;
7593                         vpp++;
7594                 }

7596                 *vec++ = (char)ret;
7597         }
7598         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7599         return (len);
7600 }

7602 /*
7603  * Statement for p_cowcnts/p_lckcnts.
7604  *
7605  * p_cowcnt is updated while mlock/munlocking MAP_PRIVATE and PROT_WRITE region
7606  * irrespective of the following factors or anything else:
7607  *
7608  *      (1) anon slots are populated or not
7609  *      (2) cow is broken or not
7610  *      (3) refcnt on ap is 1 or greater than 1
7611  *
7612  * If it's not MAP_PRIVATE and PROT_WRITE, p_lckcnt is updated during mlock
7613  * and munlock.
7614  *
7615  *
7616  * Handling p_cowcnts/p_lckcnts during copy-on-write fault:
7617  *
7618  *      if vpage has PROT_WRITE
7619  *              transfer cowcnt on the oldpage -> cowcnt on the newpage
7620  *      else
7621  *              transfer lckcnt on the oldpage -> lckcnt on the newpage
7622  *
7623  *      During copy-on-write, decrement p_cowcnt on the oldpage and increment
7624  *      p_cowcnt on the newpage *if* the corresponding vpage has PROT_WRITE.
7625  *
7626  *      We may also break COW if softlocking on read access in the physio case.
7627  *      In this case, vpage may not have PROT_WRITE. So, we need to decrement
7628  *      p_lckcnt on the oldpage and increment p_lckcnt on the newpage *if* the
7629  *      vpage doesn't have PROT_WRITE.
7630  *
7631  *
7632  * Handling p_cowcnts/p_lckcnts during mprotect on mlocked region:
7633  *
7634  *      If a MAP_PRIVATE region loses PROT_WRITE, we decrement p_cowcnt and
7635  *      increment p_lckcnt by calling page_subclaim() which takes care of
7636  *      availrmem accounting and p_lckcnt overflow.
7637  *
7638  *      If a MAP_PRIVATE region gains PROT_WRITE, we decrement p_lckcnt and
7639  *      increment p_cowcnt by calling page_addclaim() which takes care of
7640  *      availrmem availability and p_cowcnt overflow.
7641  */

7643 /*
```

```
7644  * Lock down (or unlock) pages mapped by this segment.
7645  *
7646  * XXX only creates PAGESIZE pages if anon slots are not initialized.
7647  * At fault time they will be relocated into larger pages.
7648  */
7649 static int
7650 segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
7651     int attr, int op, ulong_t *lockmap, size_t pos)
7652 {
7653         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7654         struct vpage *vpp;
7655         struct vpage *evp;
7656         page_t *pp;
7657         u_offset_t offset;
7658         u_offset_t off;
7659         int segtype;
7660         int pageprot;
7661         int claim;
7662         struct vnode *vp;
7663         ulong_t anon_index;
7664         struct anon_map *amp;
7665         struct anon *ap;
7666         struct vattr va;
7667         anon_sync_obj_t cookie;
7668         struct kshmid *sp = NULL;
7669         struct proc     *p = curproc;
7670         kproject_t      *proj = NULL;
7671         int chargeproc = 1;
7672         size_t locked_bytes = 0;
7673         size_t unlocked_bytes = 0;
7674         int err = 0;

7676         /*
7677          * Hold write lock on address space because may split or concatenate
7678          * segments
7679          */
7680         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
7680         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7682         /*
7683          * If this is a shm, use shm's project and zone, else use
7684          * project and zone of calling process
7685          */

7687         /* Determine if this segment backs a sysV shm */
7688         if (svd->amp != NULL && svd->amp->a_sp != NULL) {
7689                 ASSERT(svd->type == MAP_SHARED);
7690                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
7691                 sp = svd->amp->a_sp;
7692                 proj = sp->shm_perm.ipc_proj;
7693                 chargeproc = 0;
7694         }

7696         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
7697         if (attr) {
7698                 pageprot = attr & ~(SHARED|PRIVATE);
7699                 segtype = attr & SHARED ? MAP_SHARED : MAP_PRIVATE;

7701                 /*
7702                  * We are done if the segment types don't match
7703                  * or if we have segment level protections and
7704                  * they don't match.
7705                  */
7706                 if (svd->type != segtype) {
7707                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7708                         return (0);
```



```
7709                 }
7710                 if (svd->pageprot == 0 && svd->prot != pageprot) {
7711                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7712                         return (0);
7713                 }
7714         }

7716         if (op == MC_LOCK) {
7717                 if (svd->tr_state == SEGVN_TR_INIT) {
7718                         svd->tr_state = SEGVN_TR_OFF;
7719                 } else if (svd->tr_state == SEGVN_TR_ON) {
7720                         ASSERT(svd->amp != NULL);
7721                         segvn_textunrepl(seg, 0);
7722                         ASSERT(svd->amp == NULL &&
7723                             svd->tr_state == SEGVN_TR_OFF);
7724                 }
7725         }

7727         /*
7728          * If we're locking, then we must create a vpage structure if
7729          * none exists.  If we're unlocking, then check to see if there
7730          * is a vpage --  if not, then we could not have locked anything.
7731          */

7733         if ((vpp = svd->vpage) == NULL) {
7734                 if (op == MC_LOCK) {
7735                         segvn_vpage(seg);
7736                         if (svd->vpage == NULL) {
7737                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7738                                 return (ENOMEM);
7739                         }
7740                 } else {
7741                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7742                         return (0);
7743                 }
7744         }

7746         /*
7747          * The anonymous data vector (i.e., previously
7748          * unreferenced mapping to swap space) can be allocated
7749          * by lazily testing for its existence.
7750          */
7751         if (op == MC_LOCK && svd->amp == NULL && svd->vp == NULL) {
7752                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
7753                 svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
7754                 svd->amp->a_szc = seg->s_szc;
7755         }

7757         if ((amp = svd->amp) != NULL) {
7758                 anon_index = svd->anon_index + seg_page(seg, addr);
7759         }

7761         offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7762         evp = &svd->vpage[seg_page(seg, addr + len)];

7764         if (sp != NULL)
7765                 mutex_enter(&sp->shm_mlock);

7767         /* determine number of unlocked bytes in range for lock operation */
7768         if (op == MC_LOCK) {

7770                 if (sp == NULL) {
7771                         for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
7772                             vpp++) {
7773                                 if (!VPP_ISPPLOCK(vpp))
7774                                         unlocked_bytes += PAGESIZE;
```

```
7775                        }
7776                } else {
7777                        ulong_t         i_idx, i_edx;
7778                        anon_sync_obj_t i_cookie;
7779                        struct anon     *i_ap;
7780                        struct vnode    *i_vp;
7781                        u_offset_t      i_off;

7783                        /* Only count sysV pages once for locked memory */
7784                        i_edx = svd->anon_index + seg_page(seg, addr + len);
7785                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7786                        for (i_idx = anon_index; i_idx < i_edx; i_idx++) {
7787                                anon_array_enter(amp, i_idx, &i_cookie);
7788                                i_ap = anon_get_ptr(amp->ahp, i_idx);
7789                                if (i_ap == NULL) {
7790                                        unlocked_bytes += PAGESIZE;
7791                                        anon_array_exit(&i_cookie);
7792                                        continue;
7793                                }
7794                                swap_xlate(i_ap, &i_vp, &i_off);
7795                                anon_array_exit(&i_cookie);
7796                                pp = page_lookup(i_vp, i_off, SE_SHARED);
7797                                if (pp == NULL) {
7798                                        unlocked_bytes += PAGESIZE;
7799                                        continue;
7800                                } else if (pp->p_lckcnt == 0)
7801                                        unlocked_bytes += PAGESIZE;
7802                                page_unlock(pp);
7803                        }
7804                        ANON_LOCK_EXIT(&amp->a_rwlock);
7805                }

7807                mutex_enter(&p->p_lock);
7808                err = rctl_incr_locked_mem(p, proj, unlocked_bytes,
7809                    chargeproc);
7810                mutex_exit(&p->p_lock);

7812                if (err) {
7813                        if (sp != NULL)
7814                                mutex_exit(&sp->shm_mlock);
7815                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7816                        return (err);
7817                }
7818        }
7819        /*
7820         * Loop over all pages in the range.  Process if we're locking and
7821         * page has not already been locked in this mapping; or if we're
7822         * unlocking and the page has been locked.
7823         */
7824        for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
7825            vpp++, pos++, addr += PAGESIZE, offset += PAGESIZE, anon_index++) {
7826                if ((attr == 0 || VPP_PROT(vpp) == pageprot) &&
7827                    ((op == MC_LOCK && !VPP_ISPPLOCK(vpp)) ||
7828                    (op == MC_UNLOCK && VPP_ISPPLOCK(vpp)))) {

7830                        if (amp != NULL)
7831                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7832                        /*
7833                         * If this isn't a MAP_NORESERVE segment and
7834                         * we're locking, allocate anon slots if they
7835                         * don't exist.  The page is brought in later on.
7836                         */
7837                        if (op == MC_LOCK && svd->vp == NULL &&
7838                            ((svd->flags & MAP_NORESERVE) == 0) &&
7839                            amp != NULL &&
7840                            ((ap = anon_get_ptr(amp->ahp, anon_index))
```



```
7841                                == NULL)) {
7842                                anon_array_enter(amp, anon_index, &cookie);

7844                                if ((ap = anon_get_ptr(amp->ahp,
7845                                    anon_index)) == NULL) {
7846                                        pp = anon_zero(seg, addr, &ap,
7847                                            svd->cred);
7848                                        if (pp == NULL) {
7849                                                anon_array_exit(&cookie);
7850                                                ANON_LOCK_EXIT(&amp->a_rwlock);
7851                                                err = ENOMEM;
7852                                                goto out;
7853                                        }
7854                                        ASSERT(anon_get_ptr(amp->ahp,
7855                                            anon_index) == NULL);
7856                                        (void) anon_set_ptr(amp->ahp,
7857                                            anon_index, ap, ANON_SLEEP);
7858                                        page_unlock(pp);
7859                                }
7860                                anon_array_exit(&cookie);
7861                        }

7863                        /*
7864                         * Get name for page, accounting for
7865                         * existence of private copy.
7866                         */
7867                        ap = NULL;
7868                        if (amp != NULL) {
7869                                anon_array_enter(amp, anon_index, &cookie);
7870                                ap = anon_get_ptr(amp->ahp, anon_index);
7871                                if (ap != NULL) {
7872                                        swap_xlate(ap, &vp, &off);
7873                                } else {
7874                                        if (svd->vp == NULL &&
7875                                            (svd->flags & MAP_NORESERVE)) {
7876                                                anon_array_exit(&cookie);
7877                                                ANON_LOCK_EXIT(&amp->a_rwlock);
7878                                                continue;
7879                                        }
7880                                        vp = svd->vp;
7881                                        off = offset;
7882                                }
7883                                if (op != MC_LOCK || ap == NULL) {
7884                                        anon_array_exit(&cookie);
7885                                        ANON_LOCK_EXIT(&amp->a_rwlock);
7886                                }
7887                        } else {
7888                                vp = svd->vp;
7889                                off = offset;
7890                        }

7892                        /*
7893                         * Get page frame.  It's ok if the page is
7894                         * not available when we're unlocking, as this
7895                         * may simply mean that a page we locked got
7896                         * truncated out of existence after we locked it.
7897                         *
7898                         * Invoke VOP_GETPAGE() to obtain the page struct
7899                         * since we may need to read it from disk if its
7900                         * been paged out.
7901                         */
7902                        if (op != MC_LOCK)
7903                                pp = page_lookup(vp, off, SE_SHARED);
7904                        else {
7905                                page_t *pl[1 + 1];
7906                                int error;
```

```
7908                                    ASSERT(vp != NULL);

7910                                    error = VOP_GETPAGE(vp, (offset_t)off, PAGESIZE,
7911                                        (uint_t *)NULL, pl, PAGESIZE, seg, addr,
7912                                        S_OTHER, svd->cred, NULL);

7914                                    if (error && ap != NULL) {
7915                                            anon_array_exit(&cookie);
7916                                            ANON_LOCK_EXIT(&amp->a_rwlock);
7917                                    }

7919                                    /*
7920                                     * If the error is EDEADLK then we must bounce
7921                                     * up and drop all vm subsystem locks and then
7922                                     * retry the operation later
7923                                     * This behavior is a temporary measure because
7924                                     * ufs/sds logging is badly designed and will
7925                                     * deadlock if we don't allow this bounce to
7926                                     * happen.  The real solution is to re-design
7927                                     * the logging code to work properly.  See bug
7928                                     * 4125102 for details of the problem.
7929                                     */
7930                                    if (error == EDEADLK) {
7931                                            err = error;
7932                                            goto out;
7933                                    }
7934                                    /*
7935                                     * Quit if we fail to fault in the page.  Treat
7936                                     * the failure as an error, unless the addr
7937                                     * is mapped beyond the end of a file.
7938                                     */
7939                                    if (error && svd->vp) {
7940                                            va.va_mask = AT_SIZE;
7941                                            if (VOP_GETATTR(svd->vp, &va, 0,
7942                                                svd->cred, NULL) != 0) {
7943                                                err = EIO;
7944                                                goto out;
7945                                            }
7946                                            if (btopr(va.va_size) >=
7947                                                btopr(off + 1)) {
7948                                                err = EIO;
7949                                                goto out;
7950                                            }
7951                                            goto out;

7953                                    } else if (error) {
7954                                            err = EIO;
7955                                            goto out;
7956                                    }
7957                                    pp = pl[0];
7958                                    ASSERT(pp != NULL);
7959                            }

7961                            /*
7962                             * See Statement at the beginning of this routine.
7963                             *
7964                             * claim is always set if MAP_PRIVATE and PROT_WRITE
7965                             * irrespective of following factors:
7966                             *
7967                             * (1) anon slots are populated or not
7968                             * (2) cow is broken or not
7969                             * (3) refcnt on ap is 1 or greater than 1
7970                             *
7971                             * See 4140683 for details
7972                             */
```



```
7973                            claim = ((VPP_PROT(vpp) & PROT_WRITE) &&
7974                                (svd->type == MAP_PRIVATE));

7976                            /*
7977                             * Perform page-level operation appropriate to
7978                             * operation.  If locking, undo the SOFTLOCK
7979                             * performed to bring the page into memory
7980                             * after setting the lock.  If unlocking,
7981                             * and no page was found, account for the claim
7982                             * separately.
7983                             */
7984                            if (op == MC_LOCK) {
7985                                    int ret = 1;    /* Assume success */

7987                                    ASSERT(!VPP_ISPPLOCK(vpp));

7989                                    ret = page_pp_lock(pp, claim, 0);
7990                                    if (ap != NULL) {
7991                                            if (ap->an_pvp != NULL) {
7992                                                    anon_swap_free(ap, pp);
7993                                            }
7994                                            anon_array_exit(&cookie);
7995                                            ANON_LOCK_EXIT(&amp->a_rwlock);
7996                                    }
7997                                    if (ret == 0) {
7998                                            /* locking page failed */
7999                                            page_unlock(pp);
8000                                            err = EAGAIN;
8001                                            goto out;
8002                                    }
8003                                    VPP_SETPPLOCK(vpp);
8004                                    if (sp != NULL) {
8005                                            if (pp->p_lckcnt == 1)
8006                                                    locked_bytes += PAGESIZE;
8007                                    } else
8008                                            locked_bytes += PAGESIZE;

8010                                    if (lockmap != (ulong_t *)NULL)
8011                                            BT_SET(lockmap, pos);

8013                                    page_unlock(pp);
8014                            } else {
8015                                    ASSERT(VPP_ISPPLOCK(vpp));
8016                                    if (pp != NULL) {
8017                                            /* sysV pages should be locked */
8018                                            ASSERT(sp == NULL || pp->p_lckcnt > 0);
8019                                            page_pp_unlock(pp, claim, 0);
8020                                            if (sp != NULL) {
8021                                                    if (pp->p_lckcnt == 0)
8022                                                            unlocked_bytes
8023                                                                += PAGESIZE;
8024                                            } else
8025                                                    unlocked_bytes += PAGESIZE;
8026                                            page_unlock(pp);
8027                                    } else {
8028                                            ASSERT(sp == NULL);
8029                                            unlocked_bytes += PAGESIZE;
8030                                    }
8031                                    VPP_CLRPPLOCK(vpp);
8032                            }
8033                    }
8034            }
8035    out:
8036            if (op == MC_LOCK) {
8037                    /* Credit back bytes that did not get locked */
8038                    if ((unlocked_bytes - locked_bytes) > 0) {
```

```
8039                        if (proj == NULL)
8040                                mutex_enter(&p->p_lock);
8041                        rctl_decr_locked_mem(p, proj,
8042                            (unlocked_bytes - locked_bytes), chargeproc);
8043                        if (proj == NULL)
8044                                mutex_exit(&p->p_lock);
8045                }

8047        } else {
8048                /* Account bytes that were unlocked */
8049                if (unlocked_bytes > 0) {
8050                        if (proj == NULL)
8051                                mutex_enter(&p->p_lock);
8052                        rctl_decr_locked_mem(p, proj, unlocked_bytes,
8053                            chargeproc);
8054                        if (proj == NULL)
8055                                mutex_exit(&p->p_lock);
8056                }
8057        }
8058        if (sp != NULL)
8059                mutex_exit(&sp->shm_mlock);
8060        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

8062        return (err);
8063 }

8065 /*
8066  * Set advice from user for specified pages
8067  * There are 9 types of advice:
8068  *      MADV_NORMAL     - Normal (default) behavior (whatever that is)
8069  *      MADV_RANDOM     - Random page references
8070  *                              do not allow readahead or 'klustering'
8071  *      MADV_SEQUENTIAL - Sequential page references
8072  *                              Pages previous to the one currently being
8073  *                              accessed (determined by fault) are 'not needed'
8074  *                              and are freed immediately
8075  *      MADV_WILLNEED   - Pages are likely to be used (fault ahead in mctl)
8076  *      MADV_DONTNEED   - Pages are not needed (synced out in mctl)
8077  *      MADV_FREE       - Contents can be discarded
8078  *      MADV_ACCESS_DEFAULT- Default access
8079  *      MADV_ACCESS_LWP - Next LWP will access heavily
8080  *      MADV_ACCESS_MANY- Many LWPs or processes will access heavily
8081  */
8082 static int
8083 segvn_advise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
8084 {
8085        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8086        size_t page;
8087        int err = 0;
8088        int already_set;
8089        struct anon_map *amp;
8090        ulong_t anon_index;
8091        struct seg *next;
8092        lgrp_mem_policy_t policy;
8093        struct seg *prev;
8094        struct vnode *vp;

8096        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
8096        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

8098        /*
8099         * In case of MADV_FREE, we won't be modifying any segment private
8100         * data structures; so, we only need to grab READER's lock
8101         */
8102        if (behav != MADV_FREE) {
8103                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
```



```
8104                if (svd->tr_state != SEGVN_TR_OFF) {
8105                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8106                        return (0);
8107                }
8108        } else {
8109                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
8110        }

8112        /*
8113         * Large pages are assumed to be only turned on when accesses to the
8114         * segment's address range have spatial and temporal locality. That
8115         * justifies ignoring MADV_SEQUENTIAL for large page segments.
8116         * Also, ignore advice affecting lgroup memory allocation
8117         * if don't need to do lgroup optimizations on this system
8118         */

8120        if ((behav == MADV_SEQUENTIAL &&
8121            (seg->s_szc != 0 || HAT_IS_REGION_COOKIE_VALID(svd->rcookie))) ||
8122            (!lgrp_optimizations() && (behav == MADV_ACCESS_DEFAULT ||
8123            behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY))) {
8124                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8125                return (0);
8126        }

8128        if (behav == MADV_SEQUENTIAL || behav == MADV_ACCESS_DEFAULT ||
8129            behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY) {
8130                /*
8131                 * Since we are going to unload hat mappings
8132                 * we first have to flush the cache. Otherwise
8133                 * this might lead to system panic if another
8134                 * thread is doing physio on the range whose
8135                 * mappings are unloaded by madvise(3C).
8136                 */
8137                if (svd->softlockcnt > 0) {
8138                        /*
8139                         * If this is shared segment non 0 softlockcnt
8140                         * means locked pages are still in use.
8141                         */
8142                        if (svd->type == MAP_SHARED) {
8143                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8144                                return (EAGAIN);
8145                        }
8146                        /*
8147                         * Since we do have the segvn writers lock
8148                         * nobody can fill the cache with entries
8149                         * belonging to this seg during the purge.
8150                         * The flush either succeeds or we still
8151                         * have pending I/Os. In the later case,
8152                         * madvise(3C) fails.
8153                         */
8154                        segvn_purge(seg);
8155                        if (svd->softlockcnt > 0) {
8156                                /*
8157                                 * Since madvise(3C) is advisory and
8158                                 * it's not part of UNIX98, madvise(3C)
8159                                 * failure here doesn't cause any hardship.
8160                                 * Note that we don't block in "as" layer.
8161                                 */
8162                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8163                                return (EAGAIN);
8164                        }
8165                } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
8166                    svd->amp->a_softlockcnt > 0) {
8167                        /*
8168                         * Try to purge this amp's entries from pcache. It
8169                         * will succeed only if other segments that share the
```

```
8170                         * amp have no outstanding softlock's.
8171                         */
8172                        segvn_purge(seg);
8173                }
8174        }

8176        amp = svd->amp;
8177        vp = svd->vp;
8178        if (behav == MADV_FREE) {
8179                /*
8180                 * MADV_FREE is not supported for segments with
8181                 * underlying object; if anonmap is NULL, anon slots
8182                 * are not yet populated and there is nothing for
8183                 * us to do. As MADV_FREE is advisory, we don't
8184                 * return error in either case.
8185                 */
8186                if (vp != NULL || amp == NULL) {
8187                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8188                        return (0);
8189                }

8191                segvn_purge(seg);

8193                page = seg_page(seg, addr);
8194                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
8195                anon_disclaim(amp, svd->anon_index + page, len);
8196                ANON_LOCK_EXIT(&amp->a_rwlock);
8197                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8198                return (0);
8199        }

8201        /*
8202         * If advice is to be applied to entire segment,
8203         * use advice field in seg_data structure
8204         * otherwise use appropriate vpage entry.
8205         */
8206        if ((addr == seg->s_base) && (len == seg->s_size)) {
8207                switch (behav) {
8208                case MADV_ACCESS_LWP:
8209                case MADV_ACCESS_MANY:
8210                case MADV_ACCESS_DEFAULT:
8211                        /*
8212                         * Set memory allocation policy for this segment
8213                         */
8214                        policy = lgrp_madv_to_policy(behav, len, svd->type);
8215                        if (svd->type == MAP_SHARED)
8216                                already_set = lgrp_shm_policy_set(policy, amp,
8217                                    svd->anon_index, vp, svd->offset, len);
8218                        else {
8219                                /*
8220                                 * For private memory, need writers lock on
8221                                 * address space because the segment may be
8222                                 * split or concatenated when changing policy
8223                                 */
8224                                if (AS_READ_HELD(seg->s_as)) {
8224                                if (AS_READ_HELD(seg->s_as,
8225                                    &seg->s_as->a_lock)) {
8225                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8226                                        return (IE_RETRY);
8227                                }

8229                                already_set = lgrp_privm_policy_set(policy,
8230                                    &svd->policy_info, len);
8231                        }

8233                        /*
```



```
8234                         * If policy set already and it shouldn't be reapplied,
8235                         * don't do anything.
8236                         */
8237                        if (already_set &&
8238                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
8239                                break;

8241                        /*
8242                         * Mark any existing pages in given range for
8243                         * migration
8244                         */
8245                        page_mark_migrate(seg, addr, len, amp, svd->anon_index,
8246                            vp, svd->offset, 1);

8248                        /*
8249                         * If same policy set already or this is a shared
8250                         * memory segment, don't need to try to concatenate
8251                         * segment with adjacent ones.
8252                         */
8253                        if (already_set || svd->type == MAP_SHARED)
8254                                break;

8256                        /*
8257                         * Try to concatenate this segment with previous
8258                         * one and next one, since we changed policy for
8259                         * this one and it may be compatible with adjacent
8260                         * ones now.
8261                         */
8262                        prev = AS_SEGPREV(seg->s_as, seg);
8263                        next = AS_SEGNEXT(seg->s_as, seg);

8265                        if (next && next->s_ops == &segvn_ops &&
8266                            addr + len == next->s_base)
8267                                (void) segvn_concat(seg, next, 1);

8269                        if (prev && prev->s_ops == &segvn_ops &&
8270                            addr == prev->s_base + prev->s_size) {
8271                                /*
8272                                 * Drop lock for private data of current
8273                                 * segment before concatenating (deleting) it
8274                                 * and return IE_REATTACH to tell as_ctl() that
8275                                 * current segment has changed
8276                                 */
8277                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8278                                if (!segvn_concat(prev, seg, 1))
8279                                        err = IE_REATTACH;

8281                                return (err);
8282                        }
8283                        break;

8285                case MADV_SEQUENTIAL:
8286                        /*
8287                         * unloading mapping guarantees
8288                         * detection in segvn_fault
8289                         */
8290                        ASSERT(seg->s_szc == 0);
8291                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
8292                        hat_unload(seg->s_as->a_hat, addr, len,
8293                            HAT_UNLOAD);
8294                        /* FALLTHROUGH */
8295                case MADV_NORMAL:
8296                case MADV_RANDOM:
8297                        svd->advice = (uchar_t)behav;
8298                        svd->pageadvice = 0;
8299                        break;
```

```
8300                case MADV_WILLNEED:     /* handled in memcntl */
8301                case MADV_DONTNEED:     /* handled in memcntl */
8302                case MADV_FREE:         /* handled above */
8303                        break;
8304                default:
8305                        err = EINVAL;
8306                }
8307        } else {
8308                caddr_t                 eaddr;
8309                struct seg              *new_seg;
8310                struct segvn_data       *new_svd;
8311                u_offset_t              off;
8312                caddr_t                 oldeaddr;

8314                page = seg_page(seg, addr);

8316                segvn_vpage(seg);
8317                if (svd->vpage == NULL) {
8318                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8319                        return (ENOMEM);
8320                }

8322                switch (behav) {
8323                        struct vpage *bvpp, *evpp;

8325                case MADV_ACCESS_LWP:
8326                case MADV_ACCESS_MANY:
8327                case MADV_ACCESS_DEFAULT:
8328                        /*
8329                         * Set memory allocation policy for portion of this
8330                         * segment
8331                         */

8333                        /*
8334                         * Align address and length of advice to page
8335                         * boundaries for large pages
8336                         */
8337                        if (seg->s_szc != 0) {
8338                                size_t  pgsz;

8340                                pgsz = page_get_pagesize(seg->s_szc);
8341                                addr = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
8342                                len = P2ROUNDUP(len, pgsz);
8343                        }

8345                        /*
8346                         * Check to see whether policy is set already
8347                         */
8348                        policy = lgrp_madv_to_policy(behav, len, svd->type);

8350                        anon_index = svd->anon_index + page;
8351                        off = svd->offset + (uintptr_t)(addr - seg->s_base);

8353                        if (svd->type == MAP_SHARED)
8354                                already_set = lgrp_shm_policy_set(policy, amp,
8355                                    anon_index, vp, off, len);
8356                        else
8357                                already_set =
8358                                    (policy == svd->policy_info.mem_policy);

8360                        /*
8361                         * If policy set already and it shouldn't be reapplied,
8362                         * don't do anything.
8363                         */
8364                        if (already_set &&
8365                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
```



```
8366                                break;

8368                        /*
8369                         * For private memory, need writers lock on
8370                         * address space because the segment may be
8371                         * split or concatenated when changing policy
8372                         */
8373                        if (svd->type == MAP_PRIVATE &&
8374                            AS_READ_HELD(seg->s_as)) {
8375                            AS_READ_HELD(seg->s_as, &seg->s_as->a_lock)) {
8375                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8376                                return (IE_RETRY);
8377                        }

8379                        /*
8380                         * Mark any existing pages in given range for
8381                         * migration
8382                         */
8383                        page_mark_migrate(seg, addr, len, amp, svd->anon_index,
8384                            vp, svd->offset, 1);

8386                        /*
8387                         * Don't need to try to split or concatenate
8388                         * segments, since policy is same or this is a shared
8389                         * memory segment
8390                         */
8391                        if (already_set || svd->type == MAP_SHARED)
8392                                break;

8394                        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
8395                                ASSERT(svd->amp == NULL);
8396                                ASSERT(svd->tr_state == SEGVN_TR_OFF);
8397                                ASSERT(svd->softlockcnt == 0);
8398                                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
8399                                    HAT_REGION_TEXT);
8400                                svd->rcookie = HAT_INVALID_REGION_COOKIE;
8401                        }

8403                        /*
8404                         * Split off new segment if advice only applies to a
8405                         * portion of existing segment starting in middle
8406                         */
8407                        new_seg = NULL;
8408                        eaddr = addr + len;
8409                        oldeaddr = seg->s_base + seg->s_size;
8410                        if (addr > seg->s_base) {
8411                                /*
8412                                 * Must flush I/O page cache
8413                                 * before splitting segment
8414                                 */
8415                                if (svd->softlockcnt > 0)
8416                                        segvn_purge(seg);

8418                                /*
8419                                 * Split segment and return IE_REATTACH to tell
8420                                 * as_ctl() that current segment changed
8421                                 */
8422                                new_seg = segvn_split_seg(seg, addr);
8423                                new_svd = (struct segvn_data *)new_seg->s_data;
8424                                err = IE_REATTACH;

8426                                /*
8427                                 * If new segment ends where old one
8428                                 * did, try to concatenate the new
8429                                 * segment with next one.
8430                                 */
```

```
8431                                    if (eaddr == oldeaddr) {
8432                                            /*
8433                                             * Set policy for new segment
8434                                             */
8435                                            (void) lgrp_privm_policy_set(policy,
8436                                                &new_svd->policy_info,
8437                                                new_seg->s_size);
8438
8439                                            next = AS_SEGNEXT(new_seg->s_as,
8440                                                new_seg);
8441
8442                                            if (next &&
8443                                                next->s_ops == &segvn_ops &&
8444                                                eaddr == next->s_base)
8445                                                    (void) segvn_concat(new_seg,
8446                                                        next, 1);
8447                                    }
8448                            }
8449
8450                            /*
8451                             * Split off end of existing segment if advice only
8452                             * applies to a portion of segment ending before
8453                             * end of the existing segment
8454                             */
8455                            if (eaddr < oldeaddr) {
8456                                    /*
8457                                     * Must flush I/O page cache
8458                                     * before splitting segment
8459                                     */
8460                                    if (svd->softlockcnt > 0)
8461                                            segvn_purge(seg);
8462
8463                                    /*
8464                                     * If beginning of old segment was already
8465                                     * split off, use new segment to split end off
8466                                     * from.
8467                                     */
8468                                    if (new_seg != NULL && new_seg != seg) {
8469                                            /*
8470                                             * Split segment
8471                                             */
8472                                            (void) segvn_split_seg(new_seg, eaddr);
8473
8474                                            /*
8475                                             * Set policy for new segment
8476                                             */
8477                                            (void) lgrp_privm_policy_set(policy,
8478                                                &new_svd->policy_info,
8479                                                new_seg->s_size);
8480                                    } else {
8481                                            /*
8482                                             * Split segment and return IE_REATTACH
8483                                             * to tell as_ctl() that current
8484                                             * segment changed
8485                                             */
8486                                            (void) segvn_split_seg(seg, eaddr);
8487                                            err = IE_REATTACH;
8488
8489                                            (void) lgrp_privm_policy_set(policy,
8490                                                &svd->policy_info, seg->s_size);
8491
8492                                            /*
8493                                             * If new segment starts where old one
8494                                             * did, try to concatenate it with
8495                                             * previous segment.
8496                                             */
```



```
8497                                            if (addr == seg->s_base) {
8498                                                    prev = AS_SEGPREV(seg->s_as,
8499                                                        seg);
8500
8501                                                    /*
8502                                                     * Drop lock for private data
8503                                                     * of current segment before
8504                                                     * concatenating (deleting) it
8505                                                     */
8506                                                    if (prev &&
8507                                                        prev->s_ops ==
8508                                                        &segvn_ops &&
8509                                                        addr == prev->s_base +
8510                                                        prev->s_size) {
8511                                                            SEGVN_LOCK_EXIT(
8512                                                                seg->s_as,
8513                                                                &svd->lock);
8514                                                            (void) segvn_concat(
8515                                                                prev, seg, 1);
8516                                                            return (err);
8517                                                    }
8518                                            }
8519                                    }
8520                            }
8521                            break;
8522                    case MADV_SEQUENTIAL:
8523                            ASSERT(seg->s_szc == 0);
8524                            ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
8525                            hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
8526                            /* FALLTHROUGH */
8527                    case MADV_NORMAL:
8528                    case MADV_RANDOM:
8529                            bvpp = &svd->vpage[page];
8530                            evpp = &svd->vpage[page + (len >> PAGESHIFT)];
8531                            for (; bvpp < evpp; bvpp++)
8532                                    VPP_SETADVICE(bvpp, behav);
8533                            svd->advice = MADV_NORMAL;
8534                            break;
8535                    case MADV_WILLNEED:     /* handled in memcntl */
8536                    case MADV_DONTNEED:     /* handled in memcntl */
8537                    case MADV_FREE:         /* handled above */
8538                            break;
8539                    default:
8540                            err = EINVAL;
8541                    }
8542            }
8543            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8544            return (err);
8545    }
8546
8547    /*
8548     * There is one kind of inheritance that can be specified for pages:
8549     *
8550     *     SEGP_INH_ZERO - Pages should be zeroed in the child
8551     */
8552    static int
8553    segvn_inherit(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
8554    {
8555            struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8556            struct vpage *bvpp, *evpp;
8557            size_t page;
8558            int ret = 0;
8559
8560            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
8561            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
```

```
8562         /* Can't support something we don't know about */
8563         if (behav != SEGP_INH_ZERO)
8564                 return (ENOTSUP);

8566         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

8568         /*
8569          * This must be a straightforward anonymous segment that is mapped
8570          * privately and is not backed by a vnode.
8571          */
8572         if (svd->tr_state != SEGVN_TR_OFF ||
8573             svd->type != MAP_PRIVATE ||
8574             svd->vp != NULL) {
8575                 ret = EINVAL;
8576                 goto out;
8577         }

8579         /*
8580          * If the entire segment has been marked as inherit zero, then no reason
8581          * to do anything else.
8582          */
8583         if (svd->svn_inz == SEGVN_INZ_ALL) {
8584                 ret = 0;
8585                 goto out;
8586         }

8588         /*
8589          * If this applies to the entire segment, simply mark it and we're done.
8590          */
8591         if ((addr == seg->s_base) && (len == seg->s_size)) {
8592                 svd->svn_inz = SEGVN_INZ_ALL;
8593                 ret = 0;
8594                 goto out;
8595         }

8597         /*
8598          * We've been asked to mark a subset of this segment as inherit zero,
8599          * therefore we need to mainpulate its vpages.
8600          */
8601         if (svd->vpage == NULL) {
8602                 segvn_vpage(seg);
8603                 if (svd->vpage == NULL) {
8604                         ret = ENOMEM;
8605                         goto out;
8606                 }
8607         }

8609         svd->svn_inz = SEGVN_INZ_VPP;
8610         page = seg_page(seg, addr);
8611         bvpp = &svd->vpage[page];
8612         evpp = &svd->vpage[page + (len >> PAGESHIFT)];
8613         for (; bvpp < evpp; bvpp++)
8614                 VPP_SETINHZERO(bvpp);
8615         ret = 0;

8617 out:
8618         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8619         return (ret);
8620 }
```

unchanged_portion_omitted_

```
8736 #ifdef DEBUG
8737 static uint32_t segvn_pglock_mtbf = 0;
8738 #endif

8740 #define PCACHE_SHWLIST          ((page_t *)-2)
```



```
8741 #define NOPCACHE_SHWLIST        ((page_t *)-1)

8743 /*
8744  * Lock/Unlock anon pages over a given range. Return shadow list. This routine
8745  * uses global segment pcache to cache shadow lists (i.e. pp arrays) of pages
8746  * to avoid the overhead of per page locking, unlocking for subsequent IOs to
8747  * the same parts of the segment. Currently shadow list creation is only
8748  * supported for pure anon segments. MAP_PRIVATE segment pcache entries are
8749  * tagged with segment pointer, starting virtual address and length. This
8750  * approach for MAP_SHARED segments may add many pcache entries for the same
8751  * set of pages and lead to long hash chains that decrease pcache lookup
8752  * performance. To avoid this issue for shared segments shared anon map and
8753  * starting anon index are used for pcache entry tagging. This allows all
8754  * segments to share pcache entries for the same anon range and reduces pcache
8755  * chain's length as well as memory overhead from duplicate shadow lists and
8756  * pcache entries.
8757  *
8758  * softlockcnt field in segvn_data structure counts the number of F_SOFTLOCK'd
8759  * pages via segvn_fault() and pagelock'd pages via this routine. But pagelock
8760  * part of softlockcnt accounting is done differently for private and shared
8761  * segments. In private segment case softlock is only incremented when a new
8762  * shadow list is created but not when an existing one is found via
8763  * seg_plookup(). pcache entries have reference count incremented/decremented
8764  * by each seg_plookup()/seg_pinactive() operation. Only entries that have 0
8765  * reference count can be purged (and purging is needed before segment can be
8766  * freed). When a private segment pcache entry is purged segvn_reclaim() will
8767  * decrement softlockcnt. Since in private segment case each of its pcache
8768  * entries only belongs to this segment we can expect that when
8769  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8770  * segment purge will succeed and softlockcnt will drop to 0. In shared
8771  * segment case reference count in pcache entry counts active locks from many
8772  * different segments so we can't expect segment purging to succeed even when
8773  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8774  * segment. To be able to determine when there're no pending pagelocks in
8775  * shared segment case we don't rely on purging to make softlockcnt drop to 0
8776  * but instead softlockcnt is incremented and decremented for every
8777  * segvn_pagelock(L_PAGELOCK/L_PAGEUNLOCK) call regardless if a new shadow
8778  * list was created or an existing one was found. When softlockcnt drops to 0
8779  * this segment no longer has any claims for pcached shadow lists and the
8780  * segment can be freed even if there're still active pcache entries
8781  * shared by this segment anon map. Shared segment pcache entries belong to
8782  * anon map and are typically removed when anon map is freed after all
8783  * processes destroy the segments that use this anon map.
8784  */
8785 static int
8786 segvn_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***ppp,
8787     enum lock_type type, enum seg_rw rw)
8788 {
8789         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8790         size_t np;
8791         pgcnt_t adjustpages;
8792         pgcnt_t npages;
8793         ulong_t anon_index;
8794         uint_t protchk = (rw == S_READ) ? PROT_READ : PROT_WRITE;
8795         uint_t error;
8796         struct anon_map *amp;
8797         pgcnt_t anpgcnt;
8798         struct page **pplist, **pl, *pp;
8799         caddr_t a;
8800         size_t page;
8801         caddr_t lpgaddr, lpgeaddr;
8802         anon_sync_obj_t cookie;
8803         int anlock;
8804         struct anon_map *pamp;
8805         caddr_t paddr;
8806         seg_preclaim_cbfunc_t preclaim_callback;
```

```
8807         size_t pgsz;
8808         int use_pcache;
8809         size_t wlen;
8810         uint_t pflags = 0;
8811         int sftlck_sbase = 0;
8812         int sftlck_send = 0;

8814 #ifdef DEBUG
8815         if (type == L_PAGELOCK && segvn_pglock_mtbf) {
8816                 hrtime_t ts = gethrtime();
8817                 if ((ts % segvn_pglock_mtbf) == 0) {
8818                         return (ENOTSUP);
8819                 }
8820                 if ((ts % segvn_pglock_mtbf) == 1) {
8821                         return (EFAULT);
8822                 }
8823         }
8824 #endif

8826         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
8827             "segvn_pagelock: start seg %p addr %p", seg, addr);

8829         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
8830         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
8830         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

8832         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

8834         /*
8835          * for now we only support pagelock to anon memory. We would have to
8836          * check protections for vnode objects and call into the vnode driver.
8837          * That's too much for a fast path. Let the fault entry point handle
8838          * it.
8839          */
8840         if (svd->vp != NULL) {
8841                 if (type == L_PAGELOCK) {
8842                         error = ENOTSUP;
8843                         goto out;
8844                 }
8845                 panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
8846         }
8847         if ((amp = svd->amp) == NULL) {
8848                 if (type == L_PAGELOCK) {
8849                         error = EFAULT;
8850                         goto out;
8851                 }
8852                 panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
8853         }
8854         if (rw != S_READ && rw != S_WRITE) {
8855                 if (type == L_PAGELOCK) {
8856                         error = ENOTSUP;
8857                         goto out;
8858                 }
8859                 panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
8860         }

8862         if (seg->s_szc != 0) {
8863                 /*
8864                  * We are adjusting the pagelock region to the large page size
8865                  * boundary because the unlocked part of a large page cannot
8866                  * be freed anyway unless all constituent pages of a large
8867                  * page are locked. Bigger regions reduce pcache chain length
8868                  * and improve lookup performance. The tradeoff is that the
8869                  * very first segvn_pagelock() call for a given page is more
8870                  * expensive if only 1 page_t is needed for IO. This is only
8871                  * an issue if pcache entry doesn't get reused by several
```



```
8872                  * subsequent calls. We optimize here for the case when pcache
8873                  * is heavily used by repeated IOs to the same address range.
8874                  *
8875                  * Note segment's page size cannot change while we are holding
8876                  * as lock.  And then it cannot change while softlockcnt is
8877                  * not 0. This will allow us to correctly recalculate large
8878                  * page size region for the matching pageunlock/reclaim call
8879                  * since as_pageunlock() caller must always match
8880                  * as_pagelock() call's addr and len.
8881                  *
8882                  * For pageunlock *ppp points to the pointer of page_t that
8883                  * corresponds to the real unadjusted start address. Similar
8884                  * for pagelock *ppp must point to the pointer of page_t that
8885                  * corresponds to the real unadjusted start address.
8886                  */
8887                 pgsz = page_get_pagesize(seg->s_szc);
8888                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
8889                 adjustpages = btop((uintptr_t)(addr - lpgaddr));
8890         } else if (len < segvn_pglock_comb_thrshld) {
8891                 lpgaddr = addr;
8892                 lpgeaddr = addr + len;
8893                 adjustpages = 0;
8894                 pgsz = PAGESIZE;
8895         } else {
8896                 /*
8897                  * Align the address range of large enough requests to allow
8898                  * combining of different shadow lists into 1 to reduce memory
8899                  * overhead from potentially overlapping large shadow lists
8900                  * (worst case is we have a 1MB IO into buffers with start
8901                  * addresses separated by 4K).  Alignment is only possible if
8902                  * padded chunks have sufficient access permissions. Note
8903                  * permissions won't change between L_PAGELOCK and
8904                  * L_PAGEUNLOCK calls since non 0 softlockcnt will force
8905                  * segvn_setprot() to wait until softlockcnt drops to 0. This
8906                  * allows us to determine in L_PAGEUNLOCK the same range we
8907                  * computed in L_PAGELOCK.
8908                  *
8909                  * If alignment is limited by segment ends set
8910                  * sftlck_sbase/sftlck_send flags. In L_PAGELOCK case when
8911                  * these flags are set bump softlockcnt_sbase/softlockcnt_send
8912                  * per segment counters. In L_PAGEUNLOCK case decrease
8913                  * softlockcnt_sbase/softlockcnt_send counters if
8914                  * sftlck_sbase/sftlck_send flags are set.  When
8915                  * softlockcnt_sbase/softlockcnt_send are non 0
8916                  * segvn_concat()/segvn_extend_prev()/segvn_extend_next()
8917                  * won't merge the segments. This restriction combined with
8918                  * restriction on segment unmapping and splitting for segments
8919                  * that have non 0 softlockcnt allows L_PAGEUNLOCK to
8920                  * correctly determine the same range that was previously
8921                  * locked by matching L_PAGELOCK.
8922                  */
8923                 pflags = SEGP_PSHIFT | (segvn_pglock_comb_bshift << 16);
8924                 pgsz = PAGESIZE;
8925                 if (svd->type == MAP_PRIVATE) {
8926                         lpgaddr = (caddr_t)P2ALIGN((uintptr_t)addr,
8927                             segvn_pglock_comb_balign);
8928                         if (lpgaddr < seg->s_base) {
8929                                 lpgaddr = seg->s_base;
8930                                 sftlck_sbase = 1;
8931                         }
8932                 } else {
8933                         ulong_t aix = svd->anon_index + seg_page(seg, addr);
8934                         ulong_t aaix = P2ALIGN(aix, segvn_pglock_comb_palign);
8935                         if (aaix < svd->anon_index) {
8936                                 lpgaddr = seg->s_base;
8937                                 sftlck_sbase = 1;
```

```
8938                        } else {
8939                                lpgaddr = addr - ptob(aix - aaix);
8940                                ASSERT(lpgaddr >= seg->s_base);
8941                        }
8942                }
8943                if (svd->pageprot && lpgaddr != addr) {
8944                        struct vpage *vp = &svd->vpage[seg_page(seg, lpgaddr)];
8945                        struct vpage *evp = &svd->vpage[seg_page(seg, addr)];
8946                        while (vp < evp) {
8947                                if ((VPP_PROT(vp) & protchk) == 0) {
8948                                        break;
8949                                }
8950                                vp++;
8951                        }
8952                        if (vp < evp) {
8953                                lpgaddr = addr;
8954                                pflags = 0;
8955                        }
8956                }
8957                lpgeaddr = addr + len;
8958                if (pflags) {
8959                        if (svd->type == MAP_PRIVATE) {
8960                                lpgeaddr = (caddr_t)P2ROUNDUP(
8961                                    (uintptr_t)lpgeaddr,
8962                                    segvn_pglock_comb_balign);
8963                        } else {
8964                                ulong_t aix = svd->anon_index +
8965                                    seg_page(seg, lpgeaddr);
8966                                ulong_t aaix = P2ROUNDUP(aix,
8967                                    segvn_pglock_comb_palign);
8968                                if (aaix < aix) {
8969                                        lpgeaddr = 0;
8970                                } else {
8971                                        lpgeaddr += ptob(aaix - aix);
8972                                }
8973                        }
8974                        if (lpgeaddr == 0 ||
8975                            lpgeaddr > seg->s_base + seg->s_size) {
8976                                lpgeaddr = seg->s_base + seg->s_size;
8977                                sftlck_send = 1;
8978                        }
8979                }
8980                if (svd->pageprot && lpgeaddr != addr + len) {
8981                        struct vpage *vp;
8982                        struct vpage *evp;

8984                        vp = &svd->vpage[seg_page(seg, addr + len)];
8985                        evp = &svd->vpage[seg_page(seg, lpgeaddr)];

8987                        while (vp < evp) {
8988                                if ((VPP_PROT(vp) & protchk) == 0) {
8989                                        break;
8990                                }
8991                                vp++;
8992                        }
8993                        if (vp < evp) {
8994                                lpgeaddr = addr + len;
8995                        }
8996                }
8997                adjustpages = btop((uintptr_t)(addr - lpgaddr));
8998        }

9000        /*
9001         * For MAP_SHARED segments we create pcache entries tagged by amp and
9002         * anon index so that we can share pcache entries with other segments
9003         * that map this amp.  For private segments pcache entries are tagged
```



```
9004         * with segment and virtual address.
9005         */
9006        if (svd->type == MAP_SHARED) {
9007                pamp = amp;
9008                paddr = (caddr_t)((lpgaddr - seg->s_base) +
9009                    ptob(svd->anon_index));
9010                preclaim_callback = shamp_reclaim;
9011        } else {
9012                pamp = NULL;
9013                paddr = lpgaddr;
9014                preclaim_callback = segvn_reclaim;
9015        }

9017        if (type == L_PAGEUNLOCK) {
9018                VM_STAT_ADD(segvnvmstats.pagelock[0]);

9020                /*
9021                 * update hat ref bits for /proc. We need to make sure
9022                 * that threads tracing the ref and mod bits of the
9023                 * address space get the right data.
9024                 * Note: page ref and mod bits are updated at reclaim time
9025                 */
9026                if (seg->s_as->a_vbits) {
9027                        for (a = addr; a < addr + len; a += PAGESIZE) {
9028                                if (rw == S_WRITE) {
9029                                        hat_setstat(seg->s_as, a,
9030                                            PAGESIZE, P_REF | P_MOD);
9031                                } else {
9032                                        hat_setstat(seg->s_as, a,
9033                                            PAGESIZE, P_REF);
9034                                }
9035                        }
9036                }

9038                /*
9039                 * Check the shadow list entry after the last page used in
9040                 * this IO request. If it's NOPCACHE_SHWLIST the shadow list
9041                 * was not inserted into pcache and is not large page
9042                 * adjusted.  In this case call reclaim callback directly and
9043                 * don't adjust the shadow list start and size for large
9044                 * pages.
9045                 */
9046                npages = btop(len);
9047                if ((*ppp)[npages] == NOPCACHE_SHWLIST) {
9048                        void *ptag;
9049                        if (pamp != NULL) {
9050                                ASSERT(svd->type == MAP_SHARED);
9051                                ptag = (void *)pamp;
9052                                paddr = (caddr_t)((addr - seg->s_base) +
9053                                    ptob(svd->anon_index));
9054                        } else {
9055                                ptag = (void *)seg;
9056                                paddr = addr;
9057                        }
9058                        (*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
9059                } else {
9060                        ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
9061                            IS_SWAPFSVP((*ppp)[npages]->p_vnode));
9062                        len = lpgeaddr - lpgaddr;
9063                        npages = btop(len);
9064                        seg_pinactive(seg, pamp, paddr, len,
9065                            *ppp - adjustpages, rw, pflags, preclaim_callback);
9066                }

9068                if (pamp != NULL) {
9069                        ASSERT(svd->type == MAP_SHARED);
```

```
9070                        ASSERT(svd->softlockcnt >= npages);
9071                        atomic_add_long((ulong_t *)&svd->softlockcnt, -npages);
9072                }

9074                if (sftlck_sbase) {
9075                        ASSERT(svd->softlockcnt_sbase > 0);
9076                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_sbase);
9077                }
9078                if (sftlck_send) {
9079                        ASSERT(svd->softlockcnt_send > 0);
9080                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_send);
9081                }

9083                /*
9084                 * If someone is blocked while unmapping, we purge
9085                 * segment page cache and thus reclaim pplist synchronously
9086                 * without waiting for seg_pasync_thread. This speeds up
9087                 * unmapping in cases where munmap(2) is called, while
9088                 * raw async i/o is still in progress or where a thread
9089                 * exits on data fault in a multithreaded application.
9090                 */
9091                if (AS_ISUNMAPWAIT(seg->s_as)) {
9092                        if (svd->softlockcnt == 0) {
9093                                mutex_enter(&seg->s_as->a_contents);
9094                                if (AS_ISUNMAPWAIT(seg->s_as)) {
9095                                        AS_CLRUNMAPWAIT(seg->s_as);
9096                                        cv_broadcast(&seg->s_as->a_cv);
9097                                }
9098                                mutex_exit(&seg->s_as->a_contents);
9099                        } else if (pamp == NULL) {
9100                                /*
9101                                 * softlockcnt is not 0 and this is a
9102                                 * MAP_PRIVATE segment. Try to purge its
9103                                 * pcache entries to reduce softlockcnt.
9104                                 * If it drops to 0 segvn_reclaim()
9105                                 * will wake up a thread waiting on
9106                                 * unmapwait flag.
9107                                 *
9108                                 * We don't purge MAP_SHARED segments with non
9109                                 * 0 softlockcnt since IO is still in progress
9110                                 * for such segments.
9111                                 */
9112                                ASSERT(svd->type == MAP_PRIVATE);
9113                                segvn_purge(seg);
9114                        }
9115                }
9116                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9117                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_UNLOCK_END,
9118                    "segvn_pagelock: unlock seg %p addr %p", seg, addr);
9119                return (0);
9120        }

9122        /* The L_PAGELOCK case ... */

9124        VM_STAT_ADD(segvnvmstats.pagelock[1]);

9126        /*
9127         * For MAP_SHARED segments we have to check protections before
9128         * seg_plookup() since pcache entries may be shared by many segments
9129         * with potentially different page protections.
9130         */
9131        if (pamp != NULL) {
9132                ASSERT(svd->type == MAP_SHARED);
9133                if (svd->pageprot == 0) {
9134                        if ((svd->prot & protchk) == 0) {
9135                                error = EACCES;
```



```
9136                                goto out;
9137                        }
9138                } else {
9139                        /*
9140                         * check page protections
9141                         */
9142                        caddr_t ea;

9144                        if (seg->s_szc) {
9145                                a = lpgaddr;
9146                                ea = lpgeaddr;
9147                        } else {
9148                                a = addr;
9149                                ea = addr + len;
9150                        }
9151                        for (; a < ea; a += pgsz) {
9152                                struct vpage *vp;

9154                                ASSERT(seg->s_szc == 0 ||
9155                                    sameprot(seg, a, pgsz));
9156                                vp = &svd->vpage[seg_page(seg, a)];
9157                                if ((VPP_PROT(vp) & protchk) == 0) {
9158                                        error = EACCES;
9159                                        goto out;
9160                                }
9161                        }
9162                }
9163        }

9165        /*
9166         * try to find pages in segment page cache
9167         */
9168        pplist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
9169        if (pplist != NULL) {
9170                if (pamp != NULL) {
9171                        npages = btop((uintptr_t)(lpgeaddr - lpgaddr));
9172                        ASSERT(svd->type == MAP_SHARED);
9173                        atomic_add_long((ulong_t *)&svd->softlockcnt,
9174                            npages);
9175                }
9176                if (sftlck_sbase) {
9177                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9178                }
9179                if (sftlck_send) {
9180                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9181                }
9182                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9183                *ppp = pplist + adjustpages;
9184                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_HIT_END,
9185                    "segvn_pagelock: cache hit seg %p addr %p", seg, addr);
9186                return (0);
9187        }

9189        /*
9190         * For MAP_SHARED segments we already verified above that segment
9191         * protections allow this pagelock operation.
9192         */
9193        if (pamp == NULL) {
9194                ASSERT(svd->type == MAP_PRIVATE);
9195                if (svd->pageprot == 0) {
9196                        if ((svd->prot & protchk) == 0) {
9197                                error = EACCES;
9198                                goto out;
9199                        }
9200                        if (svd->prot & PROT_WRITE) {
9201                                wlen = lpgeaddr - lpgaddr;
```

```
9202                        } else {
9203                                wlen = 0;
9204                                ASSERT(rw == S_READ);
9205                        }
9206                } else {
9207                        int wcont = 1;
9208                        /*
9209                         * check page protections
9210                         */
9211                        for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
9212                                struct vpage *vp;

9214                                ASSERT(seg->s_szc == 0 ||
9215                                    sameprot(seg, a, pgsz));
9216                                vp = &svd->vpage[seg_page(seg, a)];
9217                                if ((VPP_PROT(vp) & protchk) == 0) {
9218                                        error = EACCES;
9219                                        goto out;
9220                                }
9221                                if (wcont && (VPP_PROT(vp) & PROT_WRITE)) {
9222                                        wlen += pgsz;
9223                                } else {
9224                                        wcont = 0;
9225                                        ASSERT(rw == S_READ);
9226                                }
9227                        }
9228                }
9229                ASSERT(rw == S_READ || wlen == lpgeaddr - lpgaddr);
9230                ASSERT(rw == S_WRITE || wlen <= lpgeaddr - lpgaddr);
9231        }

9233        /*
9234         * Only build large page adjusted shadow list if we expect to insert
9235         * it into pcache. For large enough pages it's a big overhead to
9236         * create a shadow list of the entire large page. But this overhead
9237         * should be amortized over repeated pcache hits on subsequent reuse
9238         * of this shadow list (IO into any range within this shadow list will
9239         * find it in pcache since we large page align the request for pcache
9240         * lookups). pcache performance is improved with bigger shadow lists
9241         * as it reduces the time to pcache the entire big segment and reduces
9242         * pcache chain length.
9243         */
9244        if (seg_pinsert_check(seg, pamp, paddr,
9245            lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
9246                addr = lpgaddr;
9247                len = lpgeaddr - lpgaddr;
9248                use_pcache = 1;
9249        } else {
9250                use_pcache = 0;
9251                /*
9252                 * Since this entry will not be inserted into the pcache, we
9253                 * will not do any adjustments to the starting address or
9254                 * size of the memory to be locked.
9255                 */
9256                adjustpages = 0;
9257        }
9258        npages = btop(len);

9260        pplist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
9261        pl = pplist;
9262        *ppp = pplist + adjustpages;
9263        /*
9264         * If use_pcache is 0 this shadow list is not large page adjusted.
9265         * Record this info in the last entry of shadow array so that
9266         * L_PAGEUNLOCK can determine if it should large page adjust the
9267         * address range to find the real range that was locked.
```



```
9268         */
9269        pl[npages] = use_pcache ? PCACHE_SHWLIST : NOPCACHE_SHWLIST;

9271        page = seg_page(seg, addr);
9272        anon_index = svd->anon_index + page;

9274        anlock = 0;
9275        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
9276        ASSERT(amp->a_szc >= seg->s_szc);
9277        anpgcnt = page_get_pagecnt(amp->a_szc);
9278        for (a = addr; a < addr + len; a += PAGESIZE, anon_index++) {
9279                struct anon *ap;
9280                struct vnode *vp;
9281                u_offset_t off;

9283                /*
9284                 * Lock and unlock anon array only once per large page.
9285                 * anon_array_enter() locks the root anon slot according to
9286                 * a_szc which can't change while anon map is locked.  We lock
9287                 * anon the first time through this loop and each time we
9288                 * reach anon index that corresponds to a root of a large
9289                 * page.
9290                 */
9291                if (a == addr || P2PHASE(anon_index, anpgcnt) == 0) {
9292                        ASSERT(anlock == 0);
9293                        anon_array_enter(amp, anon_index, &cookie);
9294                        anlock = 1;
9295                }
9296                ap = anon_get_ptr(amp->ahp, anon_index);

9298                /*
9299                 * We must never use seg_pcache for COW pages
9300                 * because we might end up with original page still
9301                 * lying in seg_pcache even after private page is
9302                 * created. This leads to data corruption as
9303                 * aio_write refers to the page still in cache
9304                 * while all other accesses refer to the private
9305                 * page.
9306                 */
9307                if (ap == NULL || ap->an_refcnt != 1) {
9308                        struct vpage *vpage;

9310                        if (seg->s_szc) {
9311                                error = EFAULT;
9312                                break;
9313                        }
9314                        if (svd->vpage != NULL) {
9315                                vpage = &svd->vpage[seg_page(seg, a)];
9316                        } else {
9317                                vpage = NULL;
9318                        }
9319                        ASSERT(anlock);
9320                        anon_array_exit(&cookie);
9321                        anlock = 0;
9322                        pp = NULL;
9323                        error = segvn_faultpage(seg->s_as->a_hat, seg, a, 0,
9324                            vpage, &pp, 0, F_INVAL, rw, 1);
9325                        if (error) {
9326                                error = fc_decode(error);
9327                                break;
9328                        }
9329                        anon_array_enter(amp, anon_index, &cookie);
9330                        anlock = 1;
9331                        ap = anon_get_ptr(amp->ahp, anon_index);
9332                        if (ap == NULL || ap->an_refcnt != 1) {
9333                                error = EFAULT;
```

```
9334                                    break;
9335                            }
9336                    }
9337                    swap_xlate(ap, &vp, &off);
9338                    pp = page_lookup_nowait(vp, off, SE_SHARED);
9339                    if (pp == NULL) {
9340                            error = EFAULT;
9341                            break;
9342                    }
9343                    if (ap->an_pvp != NULL) {
9344                            anon_swap_free(ap, pp);
9345                    }
9346                    /*
9347                     * Unlock anon if this is the last slot in a large page.
9348                     */
9349                    if (P2PHASE(anon_index, anpgcnt) == anpgcnt - 1) {
9350                            ASSERT(anlock);
9351                            anon_array_exit(&cookie);
9352                            anlock = 0;
9353                    }
9354                    *pplist++ = pp;
9355            }
9356            if (anlock) {           /* Ensure the lock is dropped */
9357                    anon_array_exit(&cookie);
9358            }
9359            ANON_LOCK_EXIT(&amp->a_rwlock);

9361            if (a >= addr + len) {
9362                    atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
9363                    if (pamp != NULL) {
9364                            ASSERT(svd->type == MAP_SHARED);
9365                            atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
9366                                npages);
9367                            wlen = len;
9368                    }
9369                    if (sftlck_sbase) {
9370                            atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9371                    }
9372                    if (sftlck_send) {
9373                            atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9374                    }
9375                    if (use_pcache) {
9376                            (void) seg_pinsert(seg, pamp, paddr, len, wlen, pl,
9377                                rw, pflags, preclaim_callback);
9378                    }
9379                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9380                    TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
9381                        "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
9382                    return (0);
9383            }

9385            pplist = pl;
9386            np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
9387            while (np > (uint_t)0) {
9388                    ASSERT(PAGE_LOCKED(*pplist));
9389                    page_unlock(*pplist);
9390                    np--;
9391                    pplist++;
9392            }
9393            kmem_free(pl, sizeof (page_t *) * (npages + 1));
9394    out:
9395            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9396            *ppp = NULL;
9397            TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
9398                "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
9399            return (error);
```



```
9400 }
______unchanged_portion_omitted_

9431 /*
9432  * If async argument is not 0 we are called from pcache async thread and don't
9433  * hold AS lock.
9434  */

9436 /*ARGSUSED*/
9437 static int
9438 segvn_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
9439            enum seg_rw rw, int async)
9440 {
9441            struct seg *seg = (struct seg *)ptag;
9442            struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9443            pgcnt_t np, npages;
9444            struct page **pl;

9446            npages = np = btop(len);
9447            ASSERT(npages);

9449            ASSERT(svd->vp == NULL && svd->amp != NULL);
9450            ASSERT(svd->softlockcnt >= npages);
9451            ASSERT(async || AS_LOCK_HELD(seg->s_as));
9452            ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

9453            pl = pplist;

9455            ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
9456            ASSERT(!async || pl[np] == PCACHE_SHWLIST);

9458            while (np > (uint_t)0) {
9459                    if (rw == S_WRITE) {
9460                            hat_setrefmod(*pplist);
9461                    } else {
9462                            hat_setref(*pplist);
9463                    }
9464                    page_unlock(*pplist);
9465                    np--;
9466                    pplist++;
9467            }

9469            kmem_free(pl, sizeof (page_t *) * (npages + 1));

9471            /*
9472             * If we are pcache async thread we don't hold AS lock. This means if
9473             * softlockcnt drops to 0 after the decrement below address space may
9474             * get freed. We can't allow it since after softlock derement to 0 we
9475             * still need to access as structure for possible wakeup of unmap
9476             * waiters. To prevent the disappearance of as we take this segment
9477             * segfree_syncmtx. segvn_free() also takes this mutex as a barrier to
9478             * make sure this routine completes before segment is freed.
9479             *
9480             * The second complication we have to deal with in async case is a
9481             * possibility of missed wake up of unmap wait thread. When we don't
9482             * hold as lock here we may take a_contents lock before unmap wait
9483             * thread that was first to see softlockcnt was still not 0. As a
9484             * result we'll fail to wake up an unmap wait thread. To avoid this
9485             * race we set nounmapwait flag in as structure if we drop softlockcnt
9486             * to 0 when we were called by pcache async thread.  unmapwait thread
9487             * will not block if this flag is set.
9488             */
9489            if (async) {
9490                    mutex_enter(&svd->segfree_syncmtx);
9491            }
```

```
9493         if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -npages)) {
9494                 if (async || AS_ISUNMAPWAIT(seg->s_as)) {
9495                         mutex_enter(&seg->s_as->a_contents);
9496                         if (async) {
9497                                 AS_SETNOUNMAPWAIT(seg->s_as);
9498                         }
9499                         if (AS_ISUNMAPWAIT(seg->s_as)) {
9500                                 AS_CLRUNMAPWAIT(seg->s_as);
9501                                 cv_broadcast(&seg->s_as->a_cv);
9502                         }
9503                         mutex_exit(&seg->s_as->a_contents);
9504                 }
9505         }

9507         if (async) {
9508                 mutex_exit(&svd->segfree_syncmtx);
9509         }
9510         return (0);
9511 }
```

*unchanged_portion_omitted_*

```
9700 /*
9701  * Bind text vnode segment to an amp. If we bind successfully mappings will be
9702  * established to per vnode mapping per lgroup amp pages instead of to vnode
9703  * pages. There's one amp per vnode text mapping per lgroup. Many processes
9704  * may share the same text replication amp. If a suitable amp doesn't already
9705  * exist in svntr hash table create a new one.  We may fail to bind to amp if
9706  * segment is not eligible for text replication.  Code below first checks for
9707  * these conditions. If binding is successful segment tr_state is set to on
9708  * and svd->amp points to the amp to use. Otherwise tr_state is set to off and
9709  * svd->amp remains as NULL.
9710  */
9711 static void
9712 segvn_textrepl(struct seg *seg)
9713 {
9714         struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
9715         vnode_t                 *vp = svd->vp;
9716         u_offset_t              off = svd->offset;
9717         size_t                  size = seg->s_size;
9718         u_offset_t              eoff = off + size;
9719         uint_t                  szc = seg->s_szc;
9720         ulong_t                 hash = SVNTR_HASH_FUNC(vp);
9721         svntr_t                 *svntrp;
9722         struct vattr            va;
9723         proc_t                  *p = seg->s_as->a_proc;
9724         lgrp_id_t               lgrp_id;
9725         lgrp_id_t               olid;
9726         int                     first;
9727         struct anon_map         *amp;

9729         ASSERT(AS_LOCK_HELD(seg->s_as));
9730         ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9730         ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9731         ASSERT(p != NULL);
9732         ASSERT(svd->tr_state == SEGVN_TR_INIT);
9733         ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9734         ASSERT(svd->flags & MAP_TEXT);
9735         ASSERT(svd->type == MAP_PRIVATE);
9736         ASSERT(vp != NULL && svd->amp == NULL);
9737         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
9738         ASSERT(!(svd->flags & MAP_NORESERVE) && svd->swresv == 0);
9739         ASSERT(seg->s_as != &kas);
9740         ASSERT(off < eoff);
9741         ASSERT(svntr_hashtab != NULL);

9743         /*
```



```
9744          * If numa optimizations are no longer desired bail out.
9745          */
9746         if (!lgrp_optimizations()) {
9747                 svd->tr_state = SEGVN_TR_OFF;
9748                 return;
9749         }

9751         /*
9752          * Avoid creating anon maps with size bigger than the file size.
9753          * If VOP_GETATTR() call fails bail out.
9754          */
9755         va.va_mask = AT_SIZE | AT_MTIME | AT_CTIME;
9756         if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL) != 0) {
9757                 svd->tr_state = SEGVN_TR_OFF;
9758                 SEGVN_TR_ADDSTAT(gaerr);
9759                 return;
9760         }
9761         if (btopr(va.va_size) < btopr(eoff)) {
9762                 svd->tr_state = SEGVN_TR_OFF;
9763                 SEGVN_TR_ADDSTAT(overmap);
9764                 return;
9765         }

9767         /*
9768          * VVMEXEC may not be set yet if exec() prefaults text segment. Set
9769          * this flag now before vn_is_mapped(V_WRITE) so that MAP_SHARED
9770          * mapping that checks if trcache for this vnode needs to be
9771          * invalidated can't miss us.
9772          */
9773         if (!(vp->v_flag & VVMEXEC)) {
9774                 mutex_enter(&vp->v_lock);
9775                 vp->v_flag |= VVMEXEC;
9776                 mutex_exit(&vp->v_lock);
9777         }
9778         mutex_enter(&svntr_hashtab[hash].tr_lock);
9779         /*
9780          * Bail out if potentially MAP_SHARED writable mappings exist to this
9781          * vnode.  We don't want to use old file contents from existing
9782          * replicas if this mapping was established after the original file
9783          * was changed.
9784          */
9785         if (vn_is_mapped(vp, V_WRITE)) {
9786                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9787                 svd->tr_state = SEGVN_TR_OFF;
9788                 SEGVN_TR_ADDSTAT(wrcnt);
9789                 return;
9790         }
9791         svntrp = svntr_hashtab[hash].tr_head;
9792         for (; svntrp != NULL; svntrp = svntrp->tr_next) {
9793                 ASSERT(svntrp->tr_refcnt != 0);
9794                 if (svntrp->tr_vp != vp) {
9795                         continue;
9796                 }

9798                 /*
9799                  * Bail out if the file or its attributes were changed after
9800                  * this replication entry was created since we need to use the
9801                  * latest file contents. Note that mtime test alone is not
9802                  * sufficient because a user can explicitly change mtime via
9803                  * utimes(2) interfaces back to the old value after modifiying
9804                  * the file contents. To detect this case we also have to test
9805                  * ctime which among other things records the time of the last
9806                  * mtime change by utimes(2). ctime is not changed when the file
9807                  * is only read or executed so we expect that typically existing
9808                  * replication amp's can be used most of the time.
9809                  */
```

```
9810                 if (!svntrp->tr_valid ||
9811                     svntrp->tr_mtime.tv_sec != va.va_mtime.tv_sec ||
9812                     svntrp->tr_mtime.tv_nsec != va.va_mtime.tv_nsec ||
9813                     svntrp->tr_ctime.tv_sec != va.va_ctime.tv_sec ||
9814                     svntrp->tr_ctime.tv_nsec != va.va_ctime.tv_nsec) {
9815                         mutex_exit(&svntr_hashtab[hash].tr_lock);
9816                         svd->tr_state = SEGVN_TR_OFF;
9817                         SEGVN_TR_ADDSTAT(stale);
9818                         return;
9819                 }
9820                 /*
9821                  * if off, eoff and szc match current segment we found the
9822                  * existing entry we can use.
9823                  */
9824                 if (svntrp->tr_off == off && svntrp->tr_eoff == eoff &&
9825                     svntrp->tr_szc == szc) {
9826                         break;
9827                 }
9828                 /*
9829                  * Don't create different but overlapping in file offsets
9830                  * entries to avoid replication of the same file pages more
9831                  * than once per lgroup.
9832                  */
9833                 if ((off >= svntrp->tr_off && off < svntrp->tr_eoff) ||
9834                     (eoff > svntrp->tr_off && eoff <= svntrp->tr_eoff)) {
9835                         mutex_exit(&svntr_hashtab[hash].tr_lock);
9836                         svd->tr_state = SEGVN_TR_OFF;
9837                         SEGVN_TR_ADDSTAT(overlap);
9838                         return;
9839                 }
9840         }
9841         /*
9842          * If we didn't find existing entry create a new one.
9843          */
9844         if (svntrp == NULL) {
9845                 svntrp = kmem_cache_alloc(svntr_cache, KM_NOSLEEP);
9846                 if (svntrp == NULL) {
9847                         mutex_exit(&svntr_hashtab[hash].tr_lock);
9848                         svd->tr_state = SEGVN_TR_OFF;
9849                         SEGVN_TR_ADDSTAT(nokmem);
9850                         return;
9851                 }
9852 #ifdef DEBUG
9853                 {
9854                         lgrp_id_t i;
9855                         for (i = 0; i < NLGRPS_MAX; i++) {
9856                                 ASSERT(svntrp->tr_amp[i] == NULL);
9857                         }
9858                 }
9859 #endif /* DEBUG */
9860                 svntrp->tr_vp = vp;
9861                 svntrp->tr_off = off;
9862                 svntrp->tr_eoff = eoff;
9863                 svntrp->tr_szc = szc;
9864                 svntrp->tr_valid = 1;
9865                 svntrp->tr_mtime = va.va_mtime;
9866                 svntrp->tr_ctime = va.va_ctime;
9867                 svntrp->tr_refcnt = 0;
9868                 svntrp->tr_next = svntr_hashtab[hash].tr_head;
9869                 svntr_hashtab[hash].tr_head = svntrp;
9870         }
9871         first = 1;
9872 again:
9873         /*
9874          * We want to pick a replica with pages on main thread's (t_tid = 1,
9875          * aka T1) lgrp. Currently text replication is only optimized for
```



```
9876          * workloads that either have all threads of a process on the same
9877          * lgrp or execute their large text primarily on main thread.
9878          */
9879         lgrp_id = p->p_t1_lgrpid;
9880         if (lgrp_id == LGRP_NONE) {
9881                 /*
9882                  * In case exec() prefaults text on non main thread use
9883                  * current thread lgrpid.  It will become main thread anyway
9884                  * soon.
9885                  */
9886                 lgrp_id = lgrp_home_id(curthread);
9887         }
9888         /*
9889          * Set p_tr_lgrpid to lgrpid if it hasn't been set yet.  Otherwise
9890          * just set it to NLGRPS_MAX if it's different from current process T1
9891          * home lgrp.  p_tr_lgrpid is used to detect if process uses text
9892          * replication and T1 new home is different from lgrp used for text
9893          * replication. When this happens asyncronous segvn thread rechecks if
9894          * segments should change lgrps used for text replication.  If we fail
9895          * to set p_tr_lgrpid with atomic_cas_32 then set it to NLGRPS_MAX
9896          * without cas if it's not already NLGRPS_MAX and not equal lgrp_id
9897          * we want to use.  We don't need to use cas in this case because
9898          * another thread that races in between our non atomic check and set
9899          * may only change p_tr_lgrpid to NLGRPS_MAX at this point.
9900          */
9901         ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9902         olid = p->p_tr_lgrpid;
9903         if (lgrp_id != olid && olid != NLGRPS_MAX) {
9904                 lgrp_id_t nlid = (olid == LGRP_NONE) ? lgrp_id : NLGRPS_MAX;
9905                 if (atomic_cas_32((uint32_t *)&p->p_tr_lgrpid, olid, nlid) !=
9906                     olid) {
9907                         olid = p->p_tr_lgrpid;
9908                         ASSERT(olid != LGRP_NONE);
9909                         if (olid != lgrp_id && olid != NLGRPS_MAX) {
9910                                 p->p_tr_lgrpid = NLGRPS_MAX;
9911                         }
9912                 }
9913                 ASSERT(p->p_tr_lgrpid != LGRP_NONE);
9914                 membar_producer();
9915                 /*
9916                  * lgrp_move_thread() won't schedule async recheck after
9917                  * p->p_t1_lgrpid update unless p->p_tr_lgrpid is not
9918                  * LGRP_NONE. Recheck p_t1_lgrpid once now that p->p_tr_lgrpid
9919                  * is not LGRP_NONE.
9920                  */
9921                 if (first && p->p_t1_lgrpid != LGRP_NONE &&
9922                     p->p_t1_lgrpid != lgrp_id) {
9923                         first = 0;
9924                         goto again;
9925                 }
9926         }
9927         /*
9928          * If no amp was created yet for lgrp_id create a new one as long as
9929          * we have enough memory to afford it.
9930          */
9931         if ((amp = svntrp->tr_amp[lgrp_id]) == NULL) {
9932                 size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
9933                 if (trmem > segvn_textrepl_max_bytes) {
9934                         SEGVN_TR_ADDSTAT(normem);
9935                         goto fail;
9936                 }
9937                 if (anon_try_resv_zone(size, NULL) == 0) {
9938                         SEGVN_TR_ADDSTAT(noanon);
9939                         goto fail;
9940                 }
9941                 amp = anonmap_alloc(size, size, ANON_NOSLEEP);
```

```
9942                 if (amp == NULL) {
9943                         anon_unresv_zone(size, NULL);
9944                         SEGVN_TR_ADDSTAT(nokmem);
9945                         goto fail;
9946                 }
9947                 ASSERT(amp->refcnt == 1);
9948                 amp->a_szc = szc;
9949                 svntrp->tr_amp[lgrp_id] = amp;
9950                 SEGVN_TR_ADDSTAT(newamp);
9951         }
9952         svntrp->tr_refcnt++;
9953         ASSERT(svd->svn_trnext == NULL);
9954         ASSERT(svd->svn_trprev == NULL);
9955         svd->svn_trnext = svntrp->tr_svnhead;
9956         svd->svn_trprev = NULL;
9957         if (svntrp->tr_svnhead != NULL) {
9958                 svntrp->tr_svnhead->svn_trprev = svd;
9959         }
9960         svntrp->tr_svnhead = svd;
9961         ASSERT(amp->a_szc == szc && amp->size == size && amp->swresv == size);
9962         ASSERT(amp->refcnt >= 1);
9963         svd->amp = amp;
9964         svd->anon_index = 0;
9965         svd->tr_policy_info.mem_policy = LGRP_MEM_POLICY_NEXT_SEG;
9966         svd->tr_policy_info.mem_lgrpid = lgrp_id;
9967         svd->tr_state = SEGVN_TR_ON;
9968         mutex_exit(&svntr_hashtab[hash].tr_lock);
9969         SEGVN_TR_ADDSTAT(repl);
9970         return;
9971 fail:
9972         ASSERT(segvn_textrepl_bytes >= size);
9973         atomic_add_long(&segvn_textrepl_bytes, -size);
9974         ASSERT(svntrp != NULL);
9975         ASSERT(svntrp->tr_amp[lgrp_id] == NULL);
9976         if (svntrp->tr_refcnt == 0) {
9977                 ASSERT(svntrp == svntr_hashtab[hash].tr_head);
9978                 svntr_hashtab[hash].tr_head = svntrp->tr_next;
9979                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9980                 kmem_cache_free(svntr_cache, svntrp);
9981         } else {
9982                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9983         }
9984         svd->tr_state = SEGVN_TR_OFF;
9985 }

9987 /*
9988  * Convert seg back to regular vnode mapping seg by unbinding it from its text
9989  * replication amp.  This routine is most typically called when segment is
9990  * unmapped but can also be called when segment no longer qualifies for text
9991  * replication (e.g. due to protection changes). If unload_unmap is set use
9992  * HAT_UNLOAD_UNMAP flag in hat_unload_callback().  If we are the last user of
9993  * svntr free all its anon maps and remove it from the hash table.
9994  */
9995 static void
9996 segvn_textunrepl(struct seg *seg, int unload_unmap)
9997 {
9998         struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
9999         vnode_t                 *vp = svd->vp;
10000         u_offset_t              off = svd->offset;
10001         size_t                  size = seg->s_size;
10002         u_offset_t              eoff = off + size;
10003         uint_t                  szc = seg->s_szc;
10004         ulong_t                 hash = SVNTR_HASH_FUNC(vp);
10005         svntr_t                 *svntrp;
10006         svntr_t                 **prv_svntrp;
10007         lgrp_id_t               lgrp_id = svd->tr_policy_info.mem_lgrpid;
```



```
10008         lgrp_id_t               i;

10010         ASSERT(AS_LOCK_HELD(seg->s_as));
10011         ASSERT(AS_WRITE_HELD(seg->s_as) ||
10011         ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
10012         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
10012             SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
10013         ASSERT(svd->tr_state == SEGVN_TR_ON);
10014         ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
10015         ASSERT(svd->amp != NULL);
10016         ASSERT(svd->amp->refcnt >= 1);
10017         ASSERT(svd->anon_index == 0);
10018         ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
10019         ASSERT(svntr_hashtab != NULL);

10021         mutex_enter(&svntr_hashtab[hash].tr_lock);
10022         prv_svntrp = &svntr_hashtab[hash].tr_head;
10023         for (; (svntrp = *prv_svntrp) != NULL; prv_svntrp = &svntrp->tr_next) {
10024                 ASSERT(svntrp->tr_refcnt != 0);
10025                 if (svntrp->tr_vp == vp && svntrp->tr_off == off &&
10026                     svntrp->tr_eoff == eoff && svntrp->tr_szc == szc) {
10027                         break;
10028                 }
10029         }
10030         if (svntrp == NULL) {
10031                 panic("segvn_textunrepl: svntr record not found");
10032         }
10033         if (svntrp->tr_amp[lgrp_id] != svd->amp) {
10034                 panic("segvn_textunrepl: amp mismatch");
10035         }
10036         svd->tr_state = SEGVN_TR_OFF;
10037         svd->amp = NULL;
10038         if (svd->svn_trprev == NULL) {
10039                 ASSERT(svntrp->tr_svnhead == svd);
10040                 svntrp->tr_svnhead = svd->svn_trnext;
10041                 if (svntrp->tr_svnhead != NULL) {
10042                         svntrp->tr_svnhead->svn_trprev = NULL;
10043                 }
10044                 svd->svn_trnext = NULL;
10045         } else {
10046                 svd->svn_trprev->svn_trnext = svd->svn_trnext;
10047                 if (svd->svn_trnext != NULL) {
10048                         svd->svn_trnext->svn_trprev = svd->svn_trprev;
10049                         svd->svn_trnext = NULL;
10050                 }
10051                 svd->svn_trprev = NULL;
10052         }
10053         if (--svntrp->tr_refcnt) {
10054                 mutex_exit(&svntr_hashtab[hash].tr_lock);
10055                 goto done;
10056         }
10057         *prv_svntrp = svntrp->tr_next;
10058         mutex_exit(&svntr_hashtab[hash].tr_lock);
10059         for (i = 0; i < NLGRPS_MAX; i++) {
10060                 struct anon_map *amp = svntrp->tr_amp[i];
10061                 if (amp == NULL) {
10062                         continue;
10063                 }
10064                 ASSERT(amp->refcnt == 1);
10065                 ASSERT(amp->swresv == size);
10066                 ASSERT(amp->size == size);
10067                 ASSERT(amp->a_szc == szc);
10068                 if (amp->a_szc != 0) {
10069                         anon_free_pages(amp->ahp, 0, size, szc);
10070                 } else {
10071                         anon_free(amp->ahp, 0, size);
```

```
10072                 }
10073                 svntrp->tr_amp[i] = NULL;
10074                 ASSERT(segvn_textrepl_bytes >= size);
10075                 atomic_add_long(&segvn_textrepl_bytes, -size);
10076                 anon_unresv_zone(amp->swresv, NULL);
10077                 amp->refcnt = 0;
10078                 anonmap_free(amp);
10079         }
10080         kmem_cache_free(svntr_cache, svntrp);
10081 done:
10082         hat_unload_callback(seg->s_as->a_hat, seg->s_base, size,
10083             unload_unmap ? HAT_UNLOAD_UNMAP : 0, NULL);
10084 }
_____unchanged_portion_omitted_

10188 static void
10189 segvn_trupdate_seg(struct seg *seg,
10190         segvn_data_t *svd,
10191         svntr_t *svntrp,
10192         ulong_t hash)
10193 {
10194         proc_t                  *p;
10195         lgrp_id_t               lgrp_id;
10196         struct as               *as;
10197         size_t                  size;
10198         struct anon_map         *amp;

10200         ASSERT(svd->vp != NULL);
10201         ASSERT(svd->vp == svntrp->tr_vp);
10202         ASSERT(svd->offset == svntrp->tr_off);
10203         ASSERT(svd->offset + seg->s_size == svntrp->tr_eoff);
10204         ASSERT(seg != NULL);
10205         ASSERT(svd->seg == seg);
10206         ASSERT(seg->s_data == (void *)svd);
10207         ASSERT(seg->s_szc == svntrp->tr_szc);
10208         ASSERT(svd->tr_state == SEGVN_TR_ON);
10209         ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
10210         ASSERT(svd->amp != NULL);
10211         ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
10212         ASSERT(svd->tr_policy_info.mem_lgrpid != LGRP_NONE);
10213         ASSERT(svd->tr_policy_info.mem_lgrpid < NLGRPS_MAX);
10214         ASSERT(svntrp->tr_amp[svd->tr_policy_info.mem_lgrpid] == svd->amp);
10215         ASSERT(svntrp->tr_refcnt != 0);
10216         ASSERT(mutex_owned(&svntr_hashtab[hash].tr_lock));

10218         as = seg->s_as;
10219         ASSERT(as != NULL && as != &kas);
10220         p = as->a_proc;
10221         ASSERT(p != NULL);
10222         ASSERT(p->p_tr_lgrpid != LGRP_NONE);
10223         lgrp_id = p->p_t1_lgrpid;
10224         if (lgrp_id == LGRP_NONE) {
10225                 return;
10226         }
10227         ASSERT(lgrp_id < NLGRPS_MAX);
10228         if (svd->tr_policy_info.mem_lgrpid == lgrp_id) {
10229                 return;
10230         }

10232         /*
10233          * Use tryenter locking since we are locking as/seg and svntr hash
10234          * lock in reverse from syncrounous thread order.
10235          */
10236         if (!AS_LOCK_TRYENTER(as, RW_READER)) {
10237         if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_READER)) {
                SEGVN_TR_ADDSTAT(nolock);
```



```
10238                 if (segvn_lgrp_trthr_migrs_snpsht) {
10239                         segvn_lgrp_trthr_migrs_snpsht = 0;
10240                 }
10241                 return;
10242         }
10243         if (!SEGVN_LOCK_TRYENTER(seg->s_as, &svd->lock, RW_WRITER)) {
10244                 AS_LOCK_EXIT(as);
10245                 AS_LOCK_EXIT(as, &as->a_lock);
10245                 SEGVN_TR_ADDSTAT(nolock);
10246                 if (segvn_lgrp_trthr_migrs_snpsht) {
10247                         segvn_lgrp_trthr_migrs_snpsht = 0;
10248                 }
10249                 return;
10250         }
10251         size = seg->s_size;
10252         if (svntrp->tr_amp[lgrp_id] == NULL) {
10253                 size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
10254                 if (trmem > segvn_textrepl_max_bytes) {
10255                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10256                         AS_LOCK_EXIT(as);
10257                         AS_LOCK_EXIT(as, &as->a_lock);
10257                         atomic_add_long(&segvn_textrepl_bytes, -size);
10258                         SEGVN_TR_ADDSTAT(normem);
10259                         return;
10260                 }
10261                 if (anon_try_resv_zone(size, NULL) == 0) {
10262                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10263                         AS_LOCK_EXIT(as);
10264                         AS_LOCK_EXIT(as, &as->a_lock);
10264                         atomic_add_long(&segvn_textrepl_bytes, -size);
10265                         SEGVN_TR_ADDSTAT(noanon);
10266                         return;
10267                 }
10268                 amp = anonmap_alloc(size, size, KM_NOSLEEP);
10269                 if (amp == NULL) {
10270                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10271                         AS_LOCK_EXIT(as);
10272                         AS_LOCK_EXIT(as, &as->a_lock);
10272                         atomic_add_long(&segvn_textrepl_bytes, -size);
10273                         anon_unresv_zone(size, NULL);
10274                         SEGVN_TR_ADDSTAT(nokmem);
10275                         return;
10276                 }
10277                 ASSERT(amp->refcnt == 1);
10278                 amp->a_szc = seg->s_szc;
10279                 svntrp->tr_amp[lgrp_id] = amp;
10280         }
10281         /*
10282          * We don't need to drop the bucket lock but here we give other
10283          * threads a chance.  svntr and svd can't be unlinked as long as
10284          * segment lock is held as a writer and AS held as well.  After we
10285          * retake bucket lock we'll continue from where we left. We'll be able
10286          * to reach the end of either list since new entries are always added
10287          * to the beginning of the lists.
10288          */
10289         mutex_exit(&svntr_hashtab[hash].tr_lock);
10290         hat_unload_callback(as->a_hat, seg->s_base, size, 0, NULL);
10291         mutex_enter(&svntr_hashtab[hash].tr_lock);

10293         ASSERT(svd->tr_state == SEGVN_TR_ON);
10294         ASSERT(svd->amp != NULL);
10295         ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
10296         ASSERT(svd->tr_policy_info.mem_lgrpid != lgrp_id);
10297         ASSERT(svd->amp != svntrp->tr_amp[lgrp_id]);

10299         svd->tr_policy_info.mem_lgrpid = lgrp_id;
```

```
10300         svd->amp = svntrp->tr_amp[lgrp_id];
10301         p->p_tr_lgrpid = NLGRPS_MAX;
10302         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10303         AS_LOCK_EXIT(as);
10304         AS_LOCK_EXIT(as, &as->a_lock);

10305         ASSERT(svntrp->tr_refcnt != 0);
10306         ASSERT(svd->vp == svntrp->tr_vp);
10307         ASSERT(svd->tr_policy_info.mem_lgrpid == lgrp_id);
10308         ASSERT(svd->amp != NULL && svd->amp == svntrp->tr_amp[lgrp_id]);
10309         ASSERT(svd->seg == seg);
10310         ASSERT(svd->tr_state == SEGVN_TR_ON);

10312         SEGVN_TR_ADDSTAT(asyncrepl);
10313 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   93038 Wed Nov 25 13:59:40 2015
new/usr/src/uts/common/vm/vm_as.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 346 /*
 347  * Search for the segment containing addr. If a segment containing addr
 348  * exists, that segment is returned.  If no such segment exists, and
 349  * the list spans addresses greater than addr, then the first segment
 350  * whose base is greater than addr is returned; otherwise, NULL is
 351  * returned unless tail is true, in which case the last element of the
 352  * list is returned.
 353  *
 354  * a_seglast is used to cache the last found segment for repeated
 355  * searches to the same addr (which happens frequently).
 356  */
 357 struct seg *
 358 as_findseg(struct as *as, caddr_t addr, int tail)
 359 {
 360         struct seg *seg = as->a_seglast;
 361         avl_index_t where;

 363         ASSERT(AS_LOCK_HELD(as));
 363         ASSERT(AS_LOCK_HELD(as, &as->a_lock));

 365         if (seg != NULL &&
 366             seg->s_base <= addr &&
 367             addr < seg->s_base + seg->s_size)
 368                 return (seg);

 370         seg = avl_find(&as->a_segtree, &addr, &where);
 371         if (seg != NULL)
 372                 return (as->a_seglast = seg);

 374         seg = avl_nearest(&as->a_segtree, where, AVL_AFTER);
 375         if (seg == NULL && tail)
 376                 seg = avl_last(&as->a_segtree);
 377         return (as->a_seglast = seg);
 378 }
______unchanged_portion_omitted_
 410 #endif /* VERIFY_SEGLIST */

 412 /*
 413  * Add a new segment to the address space. The avl_find()
 414  * may be expensive so we attempt to use last segment accessed
 415  * in as_gap() as an insertion point.
 416  */
 417 int
 418 as_addseg(struct as  *as, struct seg *newseg)
 419 {
 420         struct seg *seg;
 421         caddr_t addr;
 422         caddr_t eaddr;
 423         avl_index_t where;

 425         ASSERT(AS_WRITE_HELD(as));
 425         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

 427         as->a_updatedir = 1;    /* inform /proc */
 428         gethrestime(&as->a_updatetime);

 430         if (as->a_lastgaphl != NULL) {
 431                 struct seg *hseg = NULL;
 432                 struct seg *lseg = NULL;
```



```
 434                 if (as->a_lastgaphl->s_base > newseg->s_base) {
 435                         hseg = as->a_lastgaphl;
 436                         lseg = AVL_PREV(&as->a_segtree, hseg);
 437                 } else {
 438                         lseg = as->a_lastgaphl;
 439                         hseg = AVL_NEXT(&as->a_segtree, lseg);
 440                 }

 442                 if (hseg && lseg && lseg->s_base < newseg->s_base &&
 443                     hseg->s_base > newseg->s_base) {
 444                         avl_insert_here(&as->a_segtree, newseg, lseg,
 445                             AVL_AFTER);
 446                         as->a_lastgaphl = NULL;
 447                         as->a_seglast = newseg;
 448                         return (0);
 449                 }
 450                 as->a_lastgaphl = NULL;
 451         }

 453         addr = newseg->s_base;
 454         eaddr = addr + newseg->s_size;
 455 again:

 457         seg = avl_find(&as->a_segtree, &addr, &where);

 459         if (seg == NULL)
 460                 seg = avl_nearest(&as->a_segtree, where, AVL_AFTER);

 462         if (seg == NULL)
 463                 seg = avl_last(&as->a_segtree);

 465         if (seg != NULL) {
 466                 caddr_t base = seg->s_base;

 468                 /*
 469                  * If top of seg is below the requested address, then
 470                  * the insertion point is at the end of the linked list,
 471                  * and seg points to the tail of the list.  Otherwise,
 472                  * the insertion point is immediately before seg.
 473                  */
 474                 if (base + seg->s_size > addr) {
 475                         if (addr >= base || eaddr > base) {
 476 #ifdef __sparc
 477                                 extern struct seg_ops segnf_ops;

 479                                 /*
 480                                  * no-fault segs must disappear if overlaid.
 481                                  * XXX need new segment type so
 482                                  * we don't have to check s_ops
 483                                  */
 484                                 if (seg->s_ops == &segnf_ops) {
 485                                         seg_unmap(seg);
 486                                         goto again;
 487                                 }
 488 #endif
 489                                 return (-1);    /* overlapping segment */
 490                         }
 491                 }
 492         }
 493         as->a_seglast = newseg;
 494         avl_insert(&as->a_segtree, newseg, where);

 496 #ifdef VERIFY_SEGLIST
 497         as_verify(as);
 498 #endif
```

```
 499         return (0);
 500 }

 502 struct seg *
 503 as_removeseg(struct as *as, struct seg *seg)
 504 {
 505         avl_tree_t *t;

 507         ASSERT(AS_WRITE_HELD(as));
 507         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

 509         as->a_updatedir = 1;    /* inform /proc */
 510         gethrestime(&as->a_updatetime);

 512         if (seg == NULL)
 513                 return (NULL);

 515         t = &as->a_segtree;
 516         if (as->a_seglast == seg)
 517                 as->a_seglast = NULL;
 518         as->a_lastgaphl = NULL;

 520         /*
 521          * if this segment is at an address higher than
 522          * a_lastgap, set a_lastgap to the next segment (NULL if last segment)
 523          */
 524         if (as->a_lastgap &&
 525             (seg == as->a_lastgap || seg->s_base > as->a_lastgap->s_base))
 526                 as->a_lastgap = AVL_NEXT(t, seg);

 528         /*
 529          * remove the segment from the seg tree
 530          */
 531         avl_remove(t, seg);

 533 #ifdef VERIFY_SEGLIST
 534         as_verify(as);
 535 #endif
 536         return (seg);
 537 }

 539 /*
 540  * Find a segment containing addr.
 541  */
 542 struct seg *
 543 as_segat(struct as *as, caddr_t addr)
 544 {
 545         struct seg *seg = as->a_seglast;

 547         ASSERT(AS_LOCK_HELD(as));
 547         ASSERT(AS_LOCK_HELD(as, &as->a_lock));

 549         if (seg != NULL && seg->s_base <= addr &&
 550             addr < seg->s_base + seg->s_size)
 551                 return (seg);

 553         seg = avl_find(&as->a_segtree, &addr, NULL);
 554         return (seg);
 555 }
_____unchanged_portion_omitted_

 643 /*
 644  * Allocate and initialize an address space data structure.
 645  * We call hat_alloc to allow any machine dependent
 646  * information in the hat structure to be initialized.
 647  */
```



```
 648 struct as *
 649 as_alloc(void)
 650 {
 651         struct as *as;

 653         as = kmem_cache_alloc(as_cache, KM_SLEEP);

 655         as->a_flags             = 0;
 656         as->a_vbits             = 0;
 657         as->a_hrm               = NULL;
 658         as->a_seglast           = NULL;
 659         as->a_size              = 0;
 660         as->a_resvsize          = 0;
 661         as->a_updatedir         = 0;
 662         gethrestime(&as->a_updatetime);
 663         as->a_objectdir         = NULL;
 664         as->a_sizedir           = 0;
 665         as->a_userlimit         = (caddr_t)USERLIMIT;
 666         as->a_lastgap           = NULL;
 667         as->a_lastgaphl         = NULL;
 668         as->a_callbacks         = NULL;

 670         AS_LOCK_ENTER(as, RW_WRITER);
 670         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 671         as->a_hat = hat_alloc(as);      /* create hat for default system mmu */
 672         AS_LOCK_EXIT(as);
 672         AS_LOCK_EXIT(as, &as->a_lock);

 674         as->a_xhat = NULL;

 676         return (as);
 677 }

 679 /*
 680  * Free an address space data structure.
 681  * Need to free the hat first and then
 682  * all the segments on this as and finally
 683  * the space for the as struct itself.
 684  */
 685 void
 686 as_free(struct as *as)
 687 {
 688         struct hat *hat = as->a_hat;
 689         struct seg *seg, *next;
 690         int called = 0;

 692 top:
 693         /*
 694          * Invoke ALL callbacks. as_do_callbacks will do one callback
 695          * per call, and not return (-1) until the callback has completed.
 696          * When as_do_callbacks returns zero, all callbacks have completed.
 697          */
 698         mutex_enter(&as->a_contents);
 699         while (as->a_callbacks && as_do_callbacks(as, AS_ALL_EVENT, 0, 0))
 700                 ;

 702         /* This will prevent new XHATs from attaching to as */
 703         if (!called)
 704                 AS_SETBUSY(as);
 705         mutex_exit(&as->a_contents);
 706         AS_LOCK_ENTER(as, RW_WRITER);
 706         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

 708         if (!called) {
 709                 called = 1;
 710                 hat_free_start(hat);
```

```
711                 if (as->a_xhat != NULL)
712                         xhat_free_start_all(as);
713         }
714         for (seg = AS_SEGFIRST(as); seg != NULL; seg = next) {
715                 int err;
716
717                 next = AS_SEGNEXT(as, seg);
718 retry:
719                 err = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
720                 if (err == EAGAIN) {
721                         mutex_enter(&as->a_contents);
722                         if (as->a_callbacks) {
723                                 AS_LOCK_EXIT(as);
723                                 AS_LOCK_EXIT(as, &as->a_lock);
724                         } else if (!AS_ISNOUNMAPWAIT(as)) {
725                                 /*
726                                  * Memory is currently locked. Wait for a
727                                  * cv_signal that it has been unlocked, then
728                                  * try the operation again.
729                                  */
730                                 if (AS_ISUNMAPWAIT(as) == 0)
731                                         cv_broadcast(&as->a_cv);
732                                 AS_SETUNMAPWAIT(as);
733                                 AS_LOCK_EXIT(as);
733                                 AS_LOCK_EXIT(as, &as->a_lock);
734                                 while (AS_ISUNMAPWAIT(as))
735                                         cv_wait(&as->a_cv, &as->a_contents);
736                         } else {
737                                 /*
738                                  * We may have raced with
739                                  * segvn_reclaim()/segspt_reclaim(). In this
740                                  * case clean nounmapwait flag and retry since
741                                  * softlockcnt in this segment may be already
742                                  * 0.  We don't drop as writer lock so our
743                                  * number of retries without sleeping should
744                                  * be very small. See segvn_reclaim() for
745                                  * more comments.
746                                  */
747                                 AS_CLRNOUNMAPWAIT(as);
748                                 mutex_exit(&as->a_contents);
749                                 goto retry;
750                         }
751                         mutex_exit(&as->a_contents);
752                         goto top;
753                 } else {
754                         /*
755                          * We do not expect any other error return at this
756                          * time. This is similar to an ASSERT in seg_unmap()
757                          */
758                         ASSERT(err == 0);
759                 }
760         }
761         hat_free_end(hat);
762         if (as->a_xhat != NULL)
763                 xhat_free_end_all(as);
764         AS_LOCK_EXIT(as);
764         AS_LOCK_EXIT(as, &as->a_lock);
765
766         /* /proc stuff */
767         ASSERT(avl_numnodes(&as->a_wpage) == 0);
768         if (as->a_objectdir) {
769                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
770                 as->a_objectdir = NULL;
771                 as->a_sizedir = 0;
772         }
```



```
774         /*
775          * Free the struct as back to kmem.  Assert it has no segments.
776          */
777         ASSERT(avl_numnodes(&as->a_segtree) == 0);
778         kmem_cache_free(as_cache, as);
779 }
780
781 int
782 as_dup(struct as *as, struct proc *forkedproc)
783 {
784         struct as *newas;
785         struct seg *seg, *newseg;
786         size_t  purgesize = 0;
787         int error;
788
789         AS_LOCK_ENTER(as, RW_WRITER);
789         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
790         as_clearwatch(as);
791         newas = as_alloc();
792         newas->a_userlimit = as->a_userlimit;
793         newas->a_proc = forkedproc;
794
795         AS_LOCK_ENTER(newas, RW_WRITER);
795         AS_LOCK_ENTER(newas, &newas->a_lock, RW_WRITER);
796
797         /* This will prevent new XHATs from attaching */
798         mutex_enter(&as->a_contents);
799         AS_SETBUSY(as);
800         mutex_exit(&as->a_contents);
801         mutex_enter(&newas->a_contents);
802         AS_SETBUSY(newas);
803         mutex_exit(&newas->a_contents);
804
805         (void) hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_SRD);
806
807         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
808
809                 if (seg->s_flags & S_PURGE) {
810                         purgesize += seg->s_size;
811                         continue;
812                 }
813
814                 newseg = seg_alloc(newas, seg->s_base, seg->s_size);
815                 if (newseg == NULL) {
816                         AS_LOCK_EXIT(newas);
816                         AS_LOCK_EXIT(newas, &newas->a_lock);
817                         as_setwatch(as);
818                         mutex_enter(&as->a_contents);
819                         AS_CLRBUSY(as);
820                         mutex_exit(&as->a_contents);
821                         AS_LOCK_EXIT(as);
821                         AS_LOCK_EXIT(as, &as->a_lock);
822                         as_free(newas);
823                         return (-1);
824                 }
825                 if ((error = SEGOP_DUP(seg, newseg)) != 0) {
826                         /*
827                          * We call seg_free() on the new seg
828                          * because the segment is not set up
829                          * completely; i.e. it has no ops.
830                          */
831                         as_setwatch(as);
832                         mutex_enter(&as->a_contents);
833                         AS_CLRBUSY(as);
834                         mutex_exit(&as->a_contents);
835                         AS_LOCK_EXIT(as);
```

```
835                         AS_LOCK_EXIT(as, &as->a_lock);
836                         seg_free(newseg);
837                         AS_LOCK_EXIT(newas);
837                         AS_LOCK_EXIT(newas, &newas->a_lock);
838                         as_free(newas);
839                         return (error);
840                 }
841                 newas->a_size += seg->s_size;
842         }
843         newas->a_resvsize = as->a_resvsize - purgesize;

845         error = hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_ALL);
846         if (as->a_xhat != NULL)
847                 error |= xhat_dup_all(as, newas, NULL, 0, HAT_DUP_ALL);

849         mutex_enter(&newas->a_contents);
850         AS_CLRBUSY(newas);
851         mutex_exit(&newas->a_contents);
852         AS_LOCK_EXIT(newas);
852         AS_LOCK_EXIT(newas, &newas->a_lock);

854         as_setwatch(as);
855         mutex_enter(&as->a_contents);
856         AS_CLRBUSY(as);
857         mutex_exit(&as->a_contents);
858         AS_LOCK_EXIT(as);
858         AS_LOCK_EXIT(as, &as->a_lock);
859         if (error != 0) {
860                 as_free(newas);
861                 return (error);
862         }
863         forkedproc->p_as = newas;
864         return (0);
865 }

867 /*
868  * Handle a ''fault'' at addr for size bytes.
869  */
870 faultcode_t
871 as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
872         enum fault_type type, enum seg_rw rw)
873 {
874         struct seg *seg;
875         caddr_t raddr;                  /* rounded down addr */
876         size_t rsize;                   /* rounded up size */
877         size_t ssize;
878         faultcode_t res = 0;
879         caddr_t addrsav;
880         struct seg *segsav;
881         int as_lock_held;
882         klwp_t *lwp = ttolwp(curthread);
883         int is_xhat = 0;
884         int holding_wpage = 0;
885         extern struct seg_ops   segdev_ops;



889         if (as->a_hat != hat) {
890                 /* This must be an XHAT then */
891                 is_xhat = 1;

893                 if ((type != F_INVAL) || (as == &kas))
894                         return (FC_NOSUPPORT);
895         }

897 retry:
```



```
898         if (!is_xhat) {
899                 /*
900                  * Indicate that the lwp is not to be stopped while waiting
901                  * for a pagefault.  This is to avoid deadlock while debugging
902                  * a process via /proc over NFS (in particular).
903                  */
904                 if (lwp != NULL)
905                         lwp->lwp_nostop++;

907                 /*
908                  * same length must be used when we softlock and softunlock.
909                  * We don't support softunlocking lengths less than
910                  * the original length when there is largepage support.
911                  * See seg_dev.c for more comments.
912                  */
913                 switch (type) {

915                 case F_SOFTLOCK:
916                         CPU_STATS_ADD_K(vm, softlock, 1);
917                         break;

919                 case F_SOFTUNLOCK:
920                         break;

922                 case F_PROT:
923                         CPU_STATS_ADD_K(vm, prot_fault, 1);
924                         break;

926                 case F_INVAL:
927                         CPU_STATS_ENTER_K();
928                         CPU_STATS_ADDQ(CPU, vm, as_fault, 1);
929                         if (as == &kas)
930                                 CPU_STATS_ADDQ(CPU, vm, kernel_asflt, 1);
931                         CPU_STATS_EXIT_K();
932                         break;
933                 }
934         }

936         /* Kernel probe */
937         TNF_PROBE_3(address_fault, "vm pagefault", /* CSTYLED */,
938             tnf_opaque, address,        addr,
939             tnf_fault_type,     fault_type,     type,
940             tnf_seg_access,     access,         rw);

942         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
943         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
944             (size_t)raddr;

946         /*
947          * XXX -- Don't grab the as lock for segkmap. We should grab it for
948          * correctness, but then we could be stuck holding this lock for
949          * a LONG time if the fault needs to be resolved on a slow
950          * filesystem, and then no-one will be able to exec new commands,
951          * as exec'ing requires the write lock on the as.
952          */
953         if (as == &kas && segkmap && segkmap->s_base <= raddr &&
954             raddr + size < segkmap->s_base + segkmap->s_size) {
955                 /*
956                  * if (as==&kas), this can't be XHAT: we've already returned
957                  * FC_NOSUPPORT.
958                  */
959                 seg = segkmap;
960                 as_lock_held = 0;
961         } else {
962                 AS_LOCK_ENTER(as, RW_READER);
962                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
```

```
963                 if (is_xhat && avl_numnodes(&as->a_wpage) != 0) {
964                         /*
965                          * Grab and hold the writers' lock on the as
966                          * if the fault is to a watched page.
967                          * This will keep CPUs from "peeking" at the
968                          * address range while we're temporarily boosting
969                          * the permissions for the XHAT device to
970                          * resolve the fault in the segment layer.
971                          *
972                          * We could check whether faulted address
973                          * is within a watched page and only then grab
974                          * the writer lock, but this is simpler.
975                          */
976                         AS_LOCK_EXIT(as);
977                         AS_LOCK_ENTER(as, RW_WRITER);
976                         AS_LOCK_EXIT(as, &as->a_lock);
977                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
978                 }

980                 seg = as_segat(as, raddr);
981                 if (seg == NULL) {
982                         AS_LOCK_EXIT(as);
982                         AS_LOCK_EXIT(as, &as->a_lock);
983                         if ((lwp != NULL) && (!is_xhat))
984                                 lwp->lwp_nostop--;
985                         return (FC_NOMAP);
986                 }

988                 as_lock_held = 1;
989         }

991         addrsav = raddr;
992         segsav = seg;

994         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
995                 if (raddr >= seg->s_base + seg->s_size) {
996                         seg = AS_SEGNEXT(as, seg);
997                         if (seg == NULL || raddr != seg->s_base) {
998                                 res = FC_NOMAP;
999                                 break;
1000                        }
1001                }
1002                if (raddr + rsize > seg->s_base + seg->s_size)
1003                        ssize = seg->s_base + seg->s_size - raddr;
1004                else
1005                        ssize = rsize;

1007                if (!is_xhat || (seg->s_ops != &segdev_ops)) {

1009                        if (is_xhat && avl_numnodes(&as->a_wpage) != 0 &&
1010                            pr_is_watchpage_as(raddr, rw, as)) {
1011                                /*
1012                                 * Handle watch pages.  If we're faulting on a
1013                                 * watched page from an X-hat, we have to
1014                                 * restore the original permissions while we
1015                                 * handle the fault.
1016                                 */
1017                                as_clearwatch(as);
1018                                holding_wpage = 1;
1019                        }

1021                        res = SEGOP_FAULT(hat, seg, raddr, ssize, type, rw);

1023                        /* Restore watchpoints */
1024                        if (holding_wpage) {
1025                                as_setwatch(as);
```



```
1026                                holding_wpage = 0;
1027                        }

1029                        if (res != 0)
1030                                break;
1031                } else {
1032                        /* XHAT does not support seg_dev */
1033                        res = FC_NOSUPPORT;
1034                        break;
1035                }
1036        }

1038        /*
1039         * If we were SOFTLOCKing and encountered a failure,
1040         * we must SOFTUNLOCK the range we already did. (Maybe we
1041         * should just panic if we are SOFTLOCKing or even SOFTUNLOCKing
1042         * right here...)
1043         */
1044        if (res != 0 && type == F_SOFTLOCK) {
1045                for (seg = segsav; addrsav < raddr; addrsav += ssize) {
1046                        if (addrsav >= seg->s_base + seg->s_size)
1047                                seg = AS_SEGNEXT(as, seg);
1048                        ASSERT(seg != NULL);
1049                        /*
1050                         * Now call the fault routine again to perform the
1051                         * unlock using S_OTHER instead of the rw variable
1052                         * since we never got a chance to touch the pages.
1053                         */
1054                        if (raddr > seg->s_base + seg->s_size)
1055                                ssize = seg->s_base + seg->s_size - addrsav;
1056                        else
1057                                ssize = raddr - addrsav;
1058                        (void) SEGOP_FAULT(hat, seg, addrsav, ssize,
1059                            F_SOFTUNLOCK, S_OTHER);
1060                }
1061        }
1062        if (as_lock_held)
1063                AS_LOCK_EXIT(as);
1063                AS_LOCK_EXIT(as, &as->a_lock);
1064        if ((lwp != NULL) && (!is_xhat))
1065                lwp->lwp_nostop--;

1067        /*
1068         * If the lower levels returned EDEADLK for a fault,
1069         * It means that we should retry the fault.  Let's wait
1070         * a bit also to let the deadlock causing condition clear.
1071         * This is part of a gross hack to work around a design flaw
1072         * in the ufs/sds logging code and should go away when the
1073         * logging code is re-designed to fix the problem. See bug
1074         * 4125102 for details of the problem.
1075         */
1076        if (FC_ERRNO(res) == EDEADLK) {
1077                delay(deadlk_wait);
1078                res = 0;
1079                goto retry;
1080        }
1081        return (res);
1082 }



1086 /*
1087  * Asynchronous ''fault'' at addr for size bytes.
1088  */
1089 faultcode_t
1090 as_faulta(struct as *as, caddr_t addr, size_t size)
```

```
1091 {
1092         struct seg *seg;
1093         caddr_t raddr;                  /* rounded down addr */
1094         size_t rsize;                   /* rounded up size */
1095         faultcode_t res = 0;
1096         klwp_t *lwp = ttolwp(curthread);

1098 retry:
1099         /*
1100          * Indicate that the lwp is not to be stopped while waiting
1101          * for a pagefault.  This is to avoid deadlock while debugging
1102          * a process via /proc over NFS (in particular).
1103          */
1104         if (lwp != NULL)
1105                 lwp->lwp_nostop++;

1107         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1108         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1109             (size_t)raddr;

1111         AS_LOCK_ENTER(as, RW_READER);
1111         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1112         seg = as_segat(as, raddr);
1113         if (seg == NULL) {
1114                 AS_LOCK_EXIT(as);
1114                 AS_LOCK_EXIT(as, &as->a_lock);
1115                 if (lwp != NULL)
1116                         lwp->lwp_nostop--;
1117                 return (FC_NOMAP);
1118         }

1120         for (; rsize != 0; rsize -= PAGESIZE, raddr += PAGESIZE) {
1121                 if (raddr >= seg->s_base + seg->s_size) {
1122                         seg = AS_SEGNEXT(as, seg);
1123                         if (seg == NULL || raddr != seg->s_base) {
1124                                 res = FC_NOMAP;
1125                                 break;
1126                         }
1127                 }
1128                 res = SEGOP_FAULTA(seg, raddr);
1129                 if (res != 0)
1130                         break;
1131         }
1132         AS_LOCK_EXIT(as);
1132         AS_LOCK_EXIT(as, &as->a_lock);
1133         if (lwp != NULL)
1134                 lwp->lwp_nostop--;
1135         /*
1136          * If the lower levels returned EDEADLK for a fault,
1137          * It means that we should retry the fault.  Let's wait
1138          * a bit also to let the deadlock causing condition clear.
1139          * This is part of a gross hack to work around a design flaw
1140          * in the ufs/sds logging code and should go away when the
1141          * logging code is re-designed to fix the problem. See bug
1142          * 4125102 for details of the problem.
1143          */
1144         if (FC_ERRNO(res) == EDEADLK) {
1145                 delay(deadlk_wait);
1146                 res = 0;
1147                 goto retry;
1148         }
1149         return (res);
1150 }

1152 /*
1153  * Set the virtual mapping for the interval from [addr : addr + size)
```



```
1154  * in address space 'as' to have the specified protection.
1155  * It is ok for the range to cross over several segments,
1156  * as long as they are contiguous.
1157  */
1158 int
1159 as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1160 {
1161         struct seg *seg;
1162         struct as_callback *cb;
1163         size_t ssize;
1164         caddr_t raddr;                  /* rounded down addr */
1165         size_t rsize;                   /* rounded up size */
1166         int error = 0, writer = 0;
1167         caddr_t saveraddr;
1168         size_t saversize;

1170 setprot_top:
1171         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1172         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1173             (size_t)raddr;

1175         if (raddr + rsize < raddr)              /* check for wraparound */
1176                 return (ENOMEM);

1178         saveraddr = raddr;
1179         saversize = rsize;

1181         /*
1182          * Normally we only lock the as as a reader. But
1183          * if due to setprot the segment driver needs to split
1184          * a segment it will return IE_RETRY. Therefore we re-acquire
1185          * the as lock as a writer so the segment driver can change
1186          * the seg list. Also the segment driver will return IE_RETRY
1187          * after it has changed the segment list so we therefore keep
1188          * locking as a writer. Since these opeartions should be rare
1189          * want to only lock as a writer when necessary.
1190          */
1191         if (writer || avl_numnodes(&as->a_wpage) != 0) {
1192                 AS_LOCK_ENTER(as, RW_WRITER);
1192                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1193         } else {
1194                 AS_LOCK_ENTER(as, RW_READER);
1194                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1195         }

1197         as_clearwatchprot(as, raddr, rsize);
1198         seg = as_segat(as, raddr);
1199         if (seg == NULL) {
1200                 as_setwatch(as);
1201                 AS_LOCK_EXIT(as);
1201                 AS_LOCK_EXIT(as, &as->a_lock);
1202                 return (ENOMEM);
1203         }

1205         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1206                 if (raddr >= seg->s_base + seg->s_size) {
1207                         seg = AS_SEGNEXT(as, seg);
1208                         if (seg == NULL || raddr != seg->s_base) {
1209                                 error = ENOMEM;
1210                                 break;
1211                         }
1212                 }
1213                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
1214                         ssize = seg->s_base + seg->s_size - raddr;
1215                 else
1216                         ssize = rsize;
```

```
1217 retry:
1218                 error = SEGOP_SETPROT(seg, raddr, ssize, prot);

1220                 if (error == IE_NOMEM) {
1221                         error = EAGAIN;
1222                         break;
1223                 }

1225                 if (error == IE_RETRY) {
1226                         AS_LOCK_EXIT(as);
1226                         AS_LOCK_EXIT(as, &as->a_lock);
1227                         writer = 1;
1228                         goto setprot_top;
1229                 }

1231                 if (error == EAGAIN) {
1232                         /*
1233                          * Make sure we have a_lock as writer.
1234                          */
1235                         if (writer == 0) {
1236                                 AS_LOCK_EXIT(as);
1236                                 AS_LOCK_EXIT(as, &as->a_lock);
1237                                 writer = 1;
1238                                 goto setprot_top;
1239                         }

1241                         /*
1242                          * Memory is currently locked.  It must be unlocked
1243                          * before this operation can succeed through a retry.
1244                          * The possible reasons for locked memory and
1245                          * corresponding strategies for unlocking are:
1246                          * (1) Normal I/O
1247                          *      wait for a signal that the I/O operation
1248                          *      has completed and the memory is unlocked.
1249                          * (2) Asynchronous I/O
1250                          *      The aio subsystem does not unlock pages when
1251                          *      the I/O is completed. Those pages are unlocked
1252                          *      when the application calls aiowait/aioerror.
1253                          *      So, to prevent blocking forever, cv_broadcast()
1254                          *      is done to wake up aio_cleanup_thread.
1255                          *      Subsequently, segvn_reclaim will be called, and
1256                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1257                          * (3) Long term page locking:
1258                          *      Drivers intending to have pages locked for a
1259                          *      period considerably longer than for normal I/O
1260                          *      (essentially forever) may have registered for a
1261                          *      callback so they may unlock these pages on
1262                          *      request. This is needed to allow this operation
1263                          *      to succeed. Each entry on the callback list is
1264                          *      examined. If the event or address range pertains
1265                          *      the callback is invoked (unless it already is in
1266                          *      progress). The a_contents lock must be dropped
1267                          *      before the callback, so only one callback can
1268                          *      be done at a time. Go to the top and do more
1269                          *      until zero is returned. If zero is returned,
1270                          *      either there were no callbacks for this event
1271                          *      or they were already in progress.
1272                          */
1273                         mutex_enter(&as->a_contents);
1274                         if (as->a_callbacks &&
1275                             (cb = as_find_callback(as, AS_SETPROT_EVENT,
1276                             seg->s_base, seg->s_size))) {
1277                                 AS_LOCK_EXIT(as);
1277                                 AS_LOCK_EXIT(as, &as->a_lock);
1278                                 as_execute_callback(as, cb, AS_SETPROT_EVENT);
1279                         } else if (!AS_ISNOUNMAPWAIT(as)) {
```



```
1280                                 if (AS_ISUNMAPWAIT(as) == 0)
1281                                         cv_broadcast(&as->a_cv);
1282                                 AS_SETUNMAPWAIT(as);
1283                                 AS_LOCK_EXIT(as);
1283                                 AS_LOCK_EXIT(as, &as->a_lock);
1284                                 while (AS_ISUNMAPWAIT(as))
1285                                         cv_wait(&as->a_cv, &as->a_contents);
1286                         } else {
1287                                 /*
1288                                  * We may have raced with
1289                                  * segvn_reclaim()/segspt_reclaim(). In this
1290                                  * case clean nounmapwait flag and retry since
1291                                  * softlockcnt in this segment may be already
1292                                  * 0.  We don't drop as writer lock so our
1293                                  * number of retries without sleeping should
1294                                  * be very small. See segvn_reclaim() for
1295                                  * more comments.
1296                                  */
1297                                 AS_CLRNOUNMAPWAIT(as);
1298                                 mutex_exit(&as->a_contents);
1299                                 goto retry;
1300                         }
1301                         mutex_exit(&as->a_contents);
1302                         goto setprot_top;
1303                 } else if (error != 0)
1304                         break;
1305         }
1306         if (error != 0) {
1307                 as_setwatch(as);
1308         } else {
1309                 as_setwatchprot(as, saveraddr, saversize, prot);
1310         }
1311         AS_LOCK_EXIT(as);
1311         AS_LOCK_EXIT(as, &as->a_lock);
1312         return (error);
1313 }

1315 /*
1316  * Check to make sure that the interval [addr, addr + size)
1317  * in address space 'as' has at least the specified protection.
1318  * It is ok for the range to cross over several segments, as long
1319  * as they are contiguous.
1320  */
1321 int
1322 as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1323 {
1324         struct seg *seg;
1325         size_t ssize;
1326         caddr_t raddr;                  /* rounded down addr */
1327         size_t rsize;                   /* rounded up size */
1328         int error = 0;

1330         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1331         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1332             (size_t)raddr;

1334         if (raddr + rsize < raddr)              /* check for wraparound */
1335                 return (ENOMEM);

1337         /*
1338          * This is ugly as sin...
1339          * Normally, we only acquire the address space readers lock.
1340          * However, if the address space has watchpoints present,
1341          * we must acquire the writer lock on the address space for
1342          * the benefit of as_clearwatchprot() and as_setwatchprot().
1343          */
```

```
1344         if (avl_numnodes(&as->a_wpage) != 0)
1345                 AS_LOCK_ENTER(as, RW_WRITER);
1345                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1346         else
1347                 AS_LOCK_ENTER(as, RW_READER);
1347                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1348         as_clearwatchprot(as, raddr, rsize);
1349         seg = as_segat(as, raddr);
1350         if (seg == NULL) {
1351                 as_setwatch(as);
1352                 AS_LOCK_EXIT(as);
1352                 AS_LOCK_EXIT(as, &as->a_lock);
1353                 return (ENOMEM);
1354         }

1356         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1357                 if (raddr >= seg->s_base + seg->s_size) {
1358                         seg = AS_SEGNEXT(as, seg);
1359                         if (seg == NULL || raddr != seg->s_base) {
1360                                 error = ENOMEM;
1361                                 break;
1362                         }
1363                 }
1364                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
1365                         ssize = seg->s_base + seg->s_size - raddr;
1366                 else
1367                         ssize = rsize;

1369                 error = SEGOP_CHECKPROT(seg, raddr, ssize, prot);
1370                 if (error != 0)
1371                         break;
1372         }
1373         as_setwatch(as);
1374         AS_LOCK_EXIT(as);
1374         AS_LOCK_EXIT(as, &as->a_lock);
1375         return (error);
1376 }

1378 int
1379 as_unmap(struct as *as, caddr_t addr, size_t size)
1380 {
1381         struct seg *seg, *seg_next;
1382         struct as_callback *cb;
1383         caddr_t raddr, eaddr;
1384         size_t ssize, rsize = 0;
1385         int err;

1387 top:
1388         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1389         eaddr = (caddr_t)(((uintptr_t)(addr + size) + PAGEOFFSET) &
1390             (uintptr_t)PAGEMASK);

1392         AS_LOCK_ENTER(as, RW_WRITER);
1392         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

1394         as->a_updatedir = 1;    /* inform /proc */
1395         gethrestime(&as->a_updatetime);

1397         /*
1398          * Use as_findseg to find the first segment in the range, then
1399          * step through the segments in order, following s_next.
1400          */
1401         as_clearwatchprot(as, raddr, eaddr - raddr);

1403         for (seg = as_findseg(as, raddr, 0); seg != NULL; seg = seg_next) {
1404                 if (eaddr <= seg->s_base)
```



```
1405                         break;          /* eaddr was in a gap; all done */

1407                 /* this is implied by the test above */
1408                 ASSERT(raddr < eaddr);

1410                 if (raddr < seg->s_base)
1411                         raddr = seg->s_base;    /* raddr was in a gap */

1413                 if (eaddr > (seg->s_base + seg->s_size))
1414                         ssize = seg->s_base + seg->s_size - raddr;
1415                 else
1416                         ssize = eaddr - raddr;

1418                 /*
1419                  * Save next segment pointer since seg can be
1420                  * destroyed during the segment unmap operation.
1421                  */
1422                 seg_next = AS_SEGNEXT(as, seg);

1424                 /*
1425                  * We didn't count /dev/null mappings, so ignore them here.
1426                  * We'll handle MAP_NORESERVE cases in segvn_unmap(). (Again,
1427                  * we have to do this check here while we have seg.)
1428                  */
1429                 rsize = 0;
1430                 if (!SEG_IS_DEVNULL_MAPPING(seg) &&
1431                     !SEG_IS_PARTIAL_RESV(seg))
1432                         rsize = ssize;

1434 retry:
1435                 err = SEGOP_UNMAP(seg, raddr, ssize);
1436                 if (err == EAGAIN) {
1437                         /*
1438                          * Memory is currently locked.  It must be unlocked
1439                          * before this operation can succeed through a retry.
1440                          * The possible reasons for locked memory and
1441                          * corresponding strategies for unlocking are:
1442                          * (1) Normal I/O
1443                          *      wait for a signal that the I/O operation
1444                          *      has completed and the memory is unlocked.
1445                          * (2) Asynchronous I/O
1446                          *      The aio subsystem does not unlock pages when
1447                          *      the I/O is completed. Those pages are unlocked
1448                          *      when the application calls aiowait/aioerror.
1449                          *      So, to prevent blocking forever, cv_broadcast()
1450                          *      is done to wake up aio_cleanup_thread.
1451                          *      Subsequently, segvn_reclaim will be called, and
1452                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1453                          * (3) Long term page locking:
1454                          *      Drivers intending to have pages locked for a
1455                          *      period considerably longer than for normal I/O
1456                          *      (essentially forever) may have registered for a
1457                          *      callback so they may unlock these pages on
1458                          *      request. This is needed to allow this operation
1459                          *      to succeed. Each entry on the callback list is
1460                          *      examined. If the event or address range pertains
1461                          *      the callback is invoked (unless it already is in
1462                          *      progress). The a_contents lock must be dropped
1463                          *      before the callback, so only one callback can
1464                          *      be done at a time. Go to the top and do more
1465                          *      until zero is returned. If zero is returned,
1466                          *      either there were no callbacks for this event
1467                          *      or they were already in progress.
1468                          */
1469                         mutex_enter(&as->a_contents);
1470                         if (as->a_callbacks &&
```

```
1471                            (cb = as_find_callback(as, AS_UNMAP_EVENT,
1472                            seg->s_base, seg->s_size))) {
1473                                AS_LOCK_EXIT(as);
1473                                AS_LOCK_EXIT(as, &as->a_lock);
1474                                as_execute_callback(as, cb, AS_UNMAP_EVENT);
1475                        } else if (!AS_ISNOUNMAPWAIT(as)) {
1476                                if (AS_ISUNMAPWAIT(as) == 0)
1477                                        cv_broadcast(&as->a_cv);
1478                                AS_SETUNMAPWAIT(as);
1479                                AS_LOCK_EXIT(as);
1479                                AS_LOCK_EXIT(as, &as->a_lock);
1480                                while (AS_ISUNMAPWAIT(as))
1481                                        cv_wait(&as->a_cv, &as->a_contents);
1482                        } else {
1483                                /*
1484                                 * We may have raced with
1485                                 * segvn_reclaim()/segspt_reclaim(). In this
1486                                 * case clean nounmapwait flag and retry since
1487                                 * softlockcnt in this segment may be already
1488                                 * 0.  We don't drop as writer lock so our
1489                                 * number of retries without sleeping should
1490                                 * be very small. See segvn_reclaim() for
1491                                 * more comments.
1492                                 */
1493                                AS_CLRNOUNMAPWAIT(as);
1494                                mutex_exit(&as->a_contents);
1495                                goto retry;
1496                        }
1497                        mutex_exit(&as->a_contents);
1498                        goto top;
1499                } else if (err == IE_RETRY) {
1500                        AS_LOCK_EXIT(as);
1500                        AS_LOCK_EXIT(as, &as->a_lock);
1501                        goto top;
1502                } else if (err) {
1503                        as_setwatch(as);
1504                        AS_LOCK_EXIT(as);
1504                        AS_LOCK_EXIT(as, &as->a_lock);
1505                        return (-1);
1506                }

1508                as->a_size -= ssize;
1509                if (rsize)
1510                        as->a_resvsize -= rsize;
1511                raddr += ssize;
1512        }
1513        AS_LOCK_EXIT(as);
1513        AS_LOCK_EXIT(as, &as->a_lock);
1514        return (0);
1515 }

1517 static int
1518 as_map_segvn_segs(struct as *as, caddr_t addr, size_t size, uint_t szcvec,
1519     int (*crfp)(), struct segvn_crargs *vn_a, int *segcreated)
1520 {
1521        uint_t szc;
1522        uint_t nszc;
1523        int error;
1524        caddr_t a;
1525        caddr_t eaddr;
1526        size_t segsize;
1527        struct seg *seg;
1528        size_t pgsz;
1529        int do_off = (vn_a->vp != NULL || vn_a->amp != NULL);
1530        uint_t save_szcvec;
```



```
1532        ASSERT(AS_WRITE_HELD(as));
1532        ASSERT(AS_WRITE_HELD(as, &as->a_lock));
1533        ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
1534        ASSERT(IS_P2ALIGNED(size, PAGESIZE));
1535        ASSERT(vn_a->vp == NULL || vn_a->amp == NULL);
1536        if (!do_off) {
1537                vn_a->offset = 0;
1538        }

1540        if (szcvec <= 1) {
1541                seg = seg_alloc(as, addr, size);
1542                if (seg == NULL) {
1543                        return (ENOMEM);
1544                }
1545                vn_a->szc = 0;
1546                error = (*crfp)(seg, vn_a);
1547                if (error != 0) {
1548                        seg_free(seg);
1549                } else {
1550                        as->a_size += size;
1551                        as->a_resvsize += size;
1552                }
1553                return (error);
1554        }

1556        eaddr = addr + size;
1557        save_szcvec = szcvec;
1558        szcvec >>= 1;
1559        szc = 0;
1560        nszc = 0;
1561        while (szcvec) {
1562                if ((szcvec & 0x1) == 0) {
1563                        nszc++;
1564                        szcvec >>= 1;
1565                        continue;
1566                }
1567                nszc++;
1568                pgsz = page_get_pagesize(nszc);
1569                a = (caddr_t)P2ROUNDUP((uintptr_t)addr, pgsz);
1570                if (a != addr) {
1571                        ASSERT(a < eaddr);
1572                        segsize = a - addr;
1573                        seg = seg_alloc(as, addr, segsize);
1574                        if (seg == NULL) {
1575                                return (ENOMEM);
1576                        }
1577                        vn_a->szc = szc;
1578                        error = (*crfp)(seg, vn_a);
1579                        if (error != 0) {
1580                                seg_free(seg);
1581                                return (error);
1582                        }
1583                        as->a_size += segsize;
1584                        as->a_resvsize += segsize;
1585                        *segcreated = 1;
1586                        if (do_off) {
1587                                vn_a->offset += segsize;
1588                        }
1589                        addr = a;
1590                }
1591                szc = nszc;
1592                szcvec >>= 1;
1593        }

1595        ASSERT(addr < eaddr);
1596        szcvec = save_szcvec | 1; /* add 8K pages */
```

```
1597         while (szcvec) {
1598                 a = (caddr_t)P2ALIGN((uintptr_t)eaddr, pgsz);
1599                 ASSERT(a >= addr);
1600                 if (a != addr) {
1601                         segsize = a - addr;
1602                         seg = seg_alloc(as, addr, segsize);
1603                         if (seg == NULL) {
1604                                 return (ENOMEM);
1605                         }
1606                         vn_a->szc = szc;
1607                         error = (*crfp)(seg, vn_a);
1608                         if (error != 0) {
1609                                 seg_free(seg);
1610                                 return (error);
1611                         }
1612                         as->a_size += segsize;
1613                         as->a_resvsize += segsize;
1614                         *segcreated = 1;
1615                         if (do_off) {
1616                                 vn_a->offset += segsize;
1617                         }
1618                         addr = a;
1619                 }
1620                 szcvec &= ~(1 << szc);
1621                 if (szcvec) {
1622                         szc = highbit(szcvec) - 1;
1623                         pgsz = page_get_pagesize(szc);
1624                 }
1625         }
1626         ASSERT(addr == eaddr);

1628         return (0);
1629 }

1631 static int
1632 as_map_vnsegs(struct as *as, caddr_t addr, size_t size,
1633     int (*crfp)(), struct segvn_crargs *vn_a, int *segcreated)
1634 {
1635         uint_t mapflags = vn_a->flags & (MAP_TEXT | MAP_INITDATA);
1636         int type = (vn_a->type == MAP_SHARED) ? MAPPGSZC_SHM : MAPPGSZC_PRIVM;
1637         uint_t szcvec = map_pgszcvec(addr, size, (uintptr_t)addr, mapflags,
1638             type, 0);
1639         int error;
1640         struct seg *seg;
1641         struct vattr va;
1642         u_offset_t eoff;
1643         size_t save_size = 0;
1644         extern size_t textrepl_size_thresh;

1646         ASSERT(AS_WRITE_HELD(as));
1646         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
1647         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
1648         ASSERT(IS_P2ALIGNED(size, PAGESIZE));
1649         ASSERT(vn_a->vp != NULL);
1650         ASSERT(vn_a->amp == NULL);

1652 again:
1653         if (szcvec <= 1) {
1654                 seg = seg_alloc(as, addr, size);
1655                 if (seg == NULL) {
1656                         return (ENOMEM);
1657                 }
1658                 vn_a->szc = 0;
1659                 error = (*crfp)(seg, vn_a);
1660                 if (error != 0) {
1661                         seg_free(seg);
```



```
1662                 } else {
1663                         as->a_size += size;
1664                         as->a_resvsize += size;
1665                 }
1666                 return (error);
1667         }

1669         va.va_mask = AT_SIZE;
1670         if (VOP_GETATTR(vn_a->vp, &va, ATTR_HINT, vn_a->cred, NULL) != 0) {
1671                 szcvec = 0;
1672                 goto again;
1673         }
1674         eoff = vn_a->offset & PAGEMASK;
1675         if (eoff >= va.va_size) {
1676                 szcvec = 0;
1677                 goto again;
1678         }
1679         eoff += size;
1680         if (btopr(va.va_size) < btopr(eoff)) {
1681                 save_size = size;
1682                 size = va.va_size - (vn_a->offset & PAGEMASK);
1683                 size = P2ROUNDUP_TYPED(size, PAGESIZE, size_t);
1684                 szcvec = map_pgszcvec(addr, size, (uintptr_t)addr, mapflags,
1685                     type, 0);
1686                 if (szcvec <= 1) {
1687                         size = save_size;
1688                         goto again;
1689                 }
1690         }

1692         if (size > textrepl_size_thresh) {
1693                 vn_a->flags |= _MAP_TEXTREPL;
1694         }
1695         error = as_map_segvn_segs(as, addr, size, szcvec, crfp, vn_a,
1696             segcreated);
1697         if (error != 0) {
1698                 return (error);
1699         }
1700         if (save_size) {
1701                 addr += size;
1702                 size = save_size - size;
1703                 szcvec = 0;
1704                 goto again;
1705         }
1706         return (0);
1707 }

1709 /*
1710  * as_map_ansegs: shared or private anonymous memory.  Note that the flags
1711  * passed to map_pgszvec cannot be MAP_INITDATA, for anon.
1712  */
1713 static int
1714 as_map_ansegs(struct as *as, caddr_t addr, size_t size,
1715     int (*crfp)(), struct segvn_crargs *vn_a, int *segcreated)
1716 {
1717         uint_t szcvec;
1718         uchar_t type;

1720         ASSERT(vn_a->type == MAP_SHARED || vn_a->type == MAP_PRIVATE);
1721         if (vn_a->type == MAP_SHARED) {
1722                 type = MAPPGSZC_SHM;
1723         } else if (vn_a->type == MAP_PRIVATE) {
1724                 if (vn_a->szc == AS_MAP_HEAP) {
1725                         type = MAPPGSZC_HEAP;
1726                 } else if (vn_a->szc == AS_MAP_STACK) {
1727                         type = MAPPGSZC_STACK;
```

```
1728                } else {
1729                        type = MAPPGSZC_PRIVM;
1730                }
1731        }
1732        szcvec = map_pgszcvec(addr, size, vn_a->amp == NULL ?
1733            (uintptr_t)addr : (uintptr_t)P2ROUNDUP(vn_a->offset, PAGESIZE),
1734            (vn_a->flags & MAP_TEXT), type, 0);
1735        ASSERT(AS_WRITE_HELD(as));
1735        ASSERT(AS_WRITE_HELD(as, &as->a_lock));
1736        ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
1737        ASSERT(IS_P2ALIGNED(size, PAGESIZE));
1738        ASSERT(vn_a->vp == NULL);

1740        return (as_map_segvn_segs(as, addr, size, szcvec,
1741            crfp, vn_a, segcreated));
1742 }

1744 int
1745 as_map(struct as *as, caddr_t addr, size_t size, int (*crfp)(), void *argsp)
1746 {
1747        AS_LOCK_ENTER(as, RW_WRITER);
1747        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1748        return (as_map_locked(as, addr, size, crfp, argsp));
1749 }

1751 int
1752 as_map_locked(struct as *as, caddr_t addr, size_t size, int (*crfp)(),
1753                void *argsp)
1754 {
1755        struct seg *seg = NULL;
1756        caddr_t raddr;                  /* rounded down addr */
1757        size_t rsize;                   /* rounded up size */
1758        int error;
1759        int unmap = 0;
1760        struct proc *p = curproc;
1761        struct segvn_crargs crargs;

1763        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1764        rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1765            (size_t)raddr;

1767        /*
1768         * check for wrap around
1769         */
1770        if ((raddr + rsize < raddr) || (as->a_size > (ULONG_MAX - size))) {
1771                AS_LOCK_EXIT(as);
1771                AS_LOCK_EXIT(as, &as->a_lock);
1772                return (ENOMEM);
1773        }

1775        as->a_updatedir = 1;    /* inform /proc */
1776        gethrestime(&as->a_updatetime);

1778        if (as != &kas && as->a_size + rsize > (size_t)p->p_vmem_ctl) {
1779                AS_LOCK_EXIT(as);
1779                AS_LOCK_EXIT(as, &as->a_lock);

1781                (void) rctl_action(rctlproc_legacy[RLIMIT_VMEM], p->p_rctls, p,
1782                    RCA_UNSAFE_ALL);

1784                return (ENOMEM);
1785        }

1787        if (AS_MAP_CHECK_VNODE_LPOOB(crfp, argsp)) {
1788                crargs = *(struct segvn_crargs *)argsp;
1789                error = as_map_vnsegs(as, raddr, rsize, crfp, &crargs, &unmap);
```



```
1790                if (error != 0) {
1791                        AS_LOCK_EXIT(as);
1791                        AS_LOCK_EXIT(as, &as->a_lock);
1792                        if (unmap) {
1793                                (void) as_unmap(as, addr, size);
1794                        }
1795                        return (error);
1796                }
1797        } else if (AS_MAP_CHECK_ANON_LPOOB(crfp, argsp)) {
1798                crargs = *(struct segvn_crargs *)argsp;
1799                error = as_map_ansegs(as, raddr, rsize, crfp, &crargs, &unmap);
1800                if (error != 0) {
1801                        AS_LOCK_EXIT(as);
1801                        AS_LOCK_EXIT(as, &as->a_lock);
1802                        if (unmap) {
1803                                (void) as_unmap(as, addr, size);
1804                        }
1805                        return (error);
1806                }
1807        } else {
1808                seg = seg_alloc(as, addr, size);
1809                if (seg == NULL) {
1810                        AS_LOCK_EXIT(as);
1810                        AS_LOCK_EXIT(as, &as->a_lock);
1811                        return (ENOMEM);
1812                }

1814                error = (*crfp)(seg, argsp);
1815                if (error != 0) {
1816                        seg_free(seg);
1817                        AS_LOCK_EXIT(as);
1817                        AS_LOCK_EXIT(as, &as->a_lock);
1818                        return (error);
1819                }
1820                /*
1821                 * Add size now so as_unmap will work if as_ctl fails.
1822                 */
1823                as->a_size += rsize;
1824                as->a_resvsize += rsize;
1825        }

1827        as_setwatch(as);

1829        /*
1830         * If the address space is locked,
1831         * establish memory locks for the new segment.
1832         */
1833        mutex_enter(&as->a_contents);
1834        if (AS_ISPGLCK(as)) {
1835                mutex_exit(&as->a_contents);
1836                AS_LOCK_EXIT(as);
1836                AS_LOCK_EXIT(as, &as->a_lock);
1837                error = as_ctl(as, addr, size, MC_LOCK, 0, 0, NULL, 0);
1838                if (error != 0)
1839                        (void) as_unmap(as, addr, size);
1840        } else {
1841                mutex_exit(&as->a_contents);
1842                AS_LOCK_EXIT(as);
1842                AS_LOCK_EXIT(as, &as->a_lock);
1843        }
1844        return (error);
1845 }


1848 /*
1849  * Delete all segments in the address space marked with S_PURGE.
```

```
1850  * This is currently used for Sparc V9 nofault ASI segments (seg_nf.c).
1851  * These segments are deleted as a first step before calls to as_gap(), so
1852  * that they don't affect mmap() or shmat().
1853  */
1854 void
1855 as_purge(struct as *as)
1856 {
1857         struct seg *seg;
1858         struct seg *next_seg;

1860         /*
1861          * the setting of NEEDSPURGE is protect by as_rangelock(), so
1862          * no need to grab a_contents mutex for this check
1863          */
1864         if ((as->a_flags & AS_NEEDSPURGE) == 0)
1865                 return;

1867         AS_LOCK_ENTER(as, RW_WRITER);
1867         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1868         next_seg = NULL;
1869         seg = AS_SEGFIRST(as);
1870         while (seg != NULL) {
1871                 next_seg = AS_SEGNEXT(as, seg);
1872                 if (seg->s_flags & S_PURGE)
1873                         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1874                 seg = next_seg;
1875         }
1876         AS_LOCK_EXIT(as);
1876         AS_LOCK_EXIT(as, &as->a_lock);

1878         mutex_enter(&as->a_contents);
1879         as->a_flags &= ~AS_NEEDSPURGE;
1880         mutex_exit(&as->a_contents);
1881 }

1883 /*
1884  * Find a hole within [*basep, *basep + *lenp), which contains a mappable
1885  * range of addresses at least "minlen" long, where the base of the range is
1886  * at "off" phase from an "align" boundary and there is space for a
1887  * "redzone"-sized redzone on eithe rside of the range.  Thus,
1888  * if align was 4M and off was 16k, the user wants a hole which will start
1889  * 16k into a 4M page.
1890  *
1891  * If flags specifies AH_HI, the hole will have the highest possible address
1892  * in the range.  We use the as->a_lastgap field to figure out where to
1893  * start looking for a gap.
1894  *
1895  * Otherwise, the gap will have the lowest possible address.
1896  *
1897  * If flags specifies AH_CONTAIN, the hole will contain the address addr.
1898  *
1899  * If an adequate hole is found, *basep and *lenp are set to reflect the part of
1900  * the hole that is within range, and 0 is returned. On failure, -1 is returned.
1901  *
1902  * NOTE: This routine is not correct when base+len overflows caddr_t.
1903  */
1904 int
1905 as_gap_aligned(struct as *as, size_t minlen, caddr_t *basep, size_t *lenp,
1906     uint_t flags, caddr_t addr, size_t align, size_t redzone, size_t off)
1907 {
1908         caddr_t lobound = *basep;
1909         caddr_t hibound = lobound + *lenp;
1910         struct seg *lseg, *hseg;
1911         caddr_t lo, hi;
1912         int forward;
1913         caddr_t save_base;
```



```
1914         size_t save_len;
1915         size_t save_minlen;
1916         size_t save_redzone;
1917         int fast_path = 1;

1919         save_base = *basep;
1920         save_len = *lenp;
1921         save_minlen = minlen;
1922         save_redzone = redzone;

1924         /*
1925          * For the first pass/fast_path, just add align and redzone into
1926          * minlen since if we get an allocation, we can guarantee that it
1927          * will fit the alignment and redzone requested.
1928          * This increases the chance that hibound will be adjusted to
1929          * a_lastgap->s_base which will likely allow us to find an
1930          * acceptable hole in the address space quicker.
1931          * If we can't find a hole with this fast_path, then we look for
1932          * smaller holes in which the alignment and offset may allow
1933          * the allocation to fit.
1934          */
1935         minlen += align;
1936         minlen += 2 * redzone;
1937         redzone = 0;

1939         AS_LOCK_ENTER(as, RW_READER);
1939         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1940         if (AS_SEGFIRST(as) == NULL) {
1941                 if (valid_va_range_aligned(basep, lenp, minlen, flags & AH_DIR,
1942                     align, redzone, off)) {
1943                         AS_LOCK_EXIT(as);
1943                         AS_LOCK_EXIT(as, &as->a_lock);
1944                         return (0);
1945                 } else {
1946                         AS_LOCK_EXIT(as);
1946                         AS_LOCK_EXIT(as, &as->a_lock);
1947                         *basep = save_base;
1948                         *lenp = save_len;
1949                         return (-1);
1950                 }
1951         }

1953 retry:
1954         /*
1955          * Set up to iterate over all the inter-segment holes in the given
1956          * direction.  lseg is NULL for the lowest-addressed hole and hseg is
1957          * NULL for the highest-addressed hole.  If moving backwards, we reset
1958          * sseg to denote the highest-addressed segment.
1959          */
1960         forward = (flags & AH_DIR) == AH_LO;
1961         if (forward) {
1962                 hseg = as_findseg(as, lobound, 1);
1963                 lseg = AS_SEGPREV(as, hseg);
1964         } else {

1966                 /*
1967                  * If allocating at least as much as the last allocation,
1968                  * use a_lastgap's base as a better estimate of hibound.
1969                  */
1970                 if (as->a_lastgap &&
1971                     minlen >= as->a_lastgap->s_size &&
1972                     hibound >= as->a_lastgap->s_base)
1973                         hibound = as->a_lastgap->s_base;

1975                 hseg = as_findseg(as, hibound, 1);
1976                 if (hseg->s_base + hseg->s_size < hibound) {
```

```
1977                        lseg = hseg;
1978                        hseg = NULL;
1979                } else {
1980                        lseg = AS_SEGPREV(as, hseg);
1981                }
1982        }

1984        for (;;) {
1985                /*
1986                 * Set lo and hi to the hole's boundaries.  (We should really
1987                 * use MAXADDR in place of hibound in the expression below,
1988                 * but can't express it easily; using hibound in its place is
1989                 * harmless.)
1990                 */
1991                lo = (lseg == NULL) ? 0 : lseg->s_base + lseg->s_size;
1992                hi = (hseg == NULL) ? hibound : hseg->s_base;
1993                /*
1994                 * If the iteration has moved past the interval from lobound
1995                 * to hibound it's pointless to continue.
1996                 */
1997                if ((forward && lo > hibound) || (!forward && hi < lobound))
1998                        break;
1999                else if (lo > hibound || hi < lobound)
2000                        goto cont;
2001                /*
2002                 * Candidate hole lies at least partially within the allowable
2003                 * range.  Restrict it to fall completely within that range,
2004                 * i.e., to [max(lo, lobound), min(hi, hibound)].
2005                 */
2006                if (lo < lobound)
2007                        lo = lobound;
2008                if (hi > hibound)
2009                        hi = hibound;
2010                /*
2011                 * Verify that the candidate hole is big enough and meets
2012                 * hardware constraints.  If the hole is too small, no need
2013                 * to do the further checks since they will fail.
2014                 */
2015                *basep = lo;
2016                *lenp = hi - lo;
2017                if (*lenp >= minlen && valid_va_range_aligned(basep, lenp,
2018                    minlen, forward ? AH_LO : AH_HI, align, redzone, off) &&
2019                    ((flags & AH_CONTAIN) == 0 ||
2020                    (*basep <= addr && *basep + *lenp > addr))) {
2021                        if (!forward)
2022                                as->a_lastgap = hseg;
2023                        if (hseg != NULL)
2024                                as->a_lastgaphl = hseg;
2025                        else
2026                                as->a_lastgaphl = lseg;
2027                        AS_LOCK_EXIT(as);
2027                        AS_LOCK_EXIT(as, &as->a_lock);
2028                        return (0);
2029                }
2030        cont:
2031                /*
2032                 * Move to the next hole.
2033                 */
2034                if (forward) {
2035                        lseg = hseg;
2036                        if (lseg == NULL)
2037                                break;
2038                        hseg = AS_SEGNEXT(as, hseg);
2039                } else {
2040                        hseg = lseg;
2041                        if (hseg == NULL)
```



```
2042                                break;
2043                        lseg = AS_SEGPREV(as, lseg);
2044                }
2045        }
2046        if (fast_path && (align != 0 || save_redzone != 0)) {
2047                fast_path = 0;
2048                minlen = save_minlen;
2049                redzone = save_redzone;
2050                goto retry;
2051        }
2052        *basep = save_base;
2053        *lenp = save_len;
2054        AS_LOCK_EXIT(as);
2054        AS_LOCK_EXIT(as, &as->a_lock);
2055        return (-1);
2056 }
______unchanged_portion_omitted_

2083 /*
2084  * Return the next range within [base, base + len) that is backed
2085  * with "real memory".  Skip holes and non-seg_vn segments.
2086  * We're lazy and only return one segment at a time.
2087  */
2088 int
2089 as_memory(struct as *as, caddr_t *basep, size_t *lenp)
2090 {
2091        extern struct seg_ops segspt_shmops;    /* needs a header file */
2092        struct seg *seg;
2093        caddr_t addr, eaddr;
2094        caddr_t segend;

2096        AS_LOCK_ENTER(as, RW_READER);
2096        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2098        addr = *basep;
2099        eaddr = addr + *lenp;

2101        seg = as_findseg(as, addr, 0);
2102        if (seg != NULL)
2103                addr = MAX(seg->s_base, addr);

2105        for (;;) {
2106                if (seg == NULL || addr >= eaddr || eaddr <= seg->s_base) {
2107                        AS_LOCK_EXIT(as);
2107                        AS_LOCK_EXIT(as, &as->a_lock);
2108                        return (EINVAL);
2109                }

2111                if (seg->s_ops == &segvn_ops) {
2112                        segend = seg->s_base + seg->s_size;
2113                        break;
2114                }

2116                /*
2117                 * We do ISM by looking into the private data
2118                 * to determine the real size of the segment.
2119                 */
2120                if (seg->s_ops == &segspt_shmops) {
2121                        segend = seg->s_base + spt_realsize(seg);
2122                        if (addr < segend)
2123                                break;
2124                }

2126                seg = AS_SEGNEXT(as, seg);

2128                if (seg != NULL)
```

```
2129                        addr = seg->s_base;
2130        }

2132        *basep = addr;

2134        if (segend > eaddr)
2135                *lenp = eaddr - addr;
2136        else
2137                *lenp = segend - addr;

2139        AS_LOCK_EXIT(as);
2139        AS_LOCK_EXIT(as, &as->a_lock);
2140        return (0);
2141 }

2143 /*
2144  * Swap the pages associated with the address space as out to
2145  * secondary storage, returning the number of bytes actually
2146  * swapped.
2147  *
2148  * The value returned is intended to correlate well with the process's
2149  * memory requirements.  Its usefulness for this purpose depends on
2150  * how well the segment-level routines do at returning accurate
2151  * information.
2152  */
2153 size_t
2154 as_swapout(struct as *as)
2155 {
2156        struct seg *seg;
2157        size_t swpcnt = 0;

2159        /*
2160         * Kernel-only processes have given up their address
2161         * spaces.  Of course, we shouldn't be attempting to
2162         * swap out such processes in the first place...
2163         */
2164        if (as == NULL)
2165                return (0);

2167        AS_LOCK_ENTER(as, RW_READER);
2167        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2169        /* Prevent XHATs from attaching */
2170        mutex_enter(&as->a_contents);
2171        AS_SETBUSY(as);
2172        mutex_exit(&as->a_contents);


2175        /*
2176         * Free all mapping resources associated with the address
2177         * space.  The segment-level swapout routines capitalize
2178         * on this unmapping by scavanging pages that have become
2179         * unmapped here.
2180         */
2181        hat_swapout(as->a_hat);
2182        if (as->a_xhat != NULL)
2183                xhat_swapout_all(as);

2185        mutex_enter(&as->a_contents);
2186        AS_CLRBUSY(as);
2187        mutex_exit(&as->a_contents);

2189        /*
2190         * Call the swapout routines of all segments in the address
2191         * space to do the actual work, accumulating the amount of
2192         * space reclaimed.
```



```
2193         */
2194        for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2195                struct seg_ops *ov = seg->s_ops;

2197                /*
2198                 * We have to check to see if the seg has
2199                 * an ops vector because the seg may have
2200                 * been in the middle of being set up when
2201                 * the process was picked for swapout.
2202                 */
2203                if ((ov != NULL) && (ov->swapout != NULL))
2204                        swpcnt += SEGOP_SWAPOUT(seg);
2205        }
2206        AS_LOCK_EXIT(as);
2206        AS_LOCK_EXIT(as, &as->a_lock);
2207        return (swpcnt);
2208 }

2210 /*
2211  * Determine whether data from the mappings in interval [addr, addr + size)
2212  * are in the primary memory (core) cache.
2213  */
2214 int
2215 as_incore(struct as *as, caddr_t addr,
2216     size_t size, char *vec, size_t *sizep)
2217 {
2218        struct seg *seg;
2219        size_t ssize;
2220        caddr_t raddr;          /* rounded down addr */
2221        size_t rsize;           /* rounded up size */
2222        size_t isize;                   /* iteration size */
2223        int error = 0;          /* result, assume success */

2225        *sizep = 0;
2226        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2227        rsize = ((((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
2228            (size_t)raddr;

2230        if (raddr + rsize < raddr)              /* check for wraparound */
2231                return (ENOMEM);

2233        AS_LOCK_ENTER(as, RW_READER);
2233        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2234        seg = as_segat(as, raddr);
2235        if (seg == NULL) {
2236                AS_LOCK_EXIT(as);
2236                AS_LOCK_EXIT(as, &as->a_lock);
2237                return (-1);
2238        }

2240        for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2241                if (raddr >= seg->s_base + seg->s_size) {
2242                        seg = AS_SEGNEXT(as, seg);
2243                        if (seg == NULL || raddr != seg->s_base) {
2244                                error = -1;
2245                                break;
2246                        }
2247                }
2248                if ((raddr + rsize) > (seg->s_base + seg->s_size))
2249                        ssize = seg->s_base + seg->s_size - raddr;
2250                else
2251                        ssize = rsize;
2252                *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2253                if (isize != ssize) {
2254                        error = -1;
2255                        break;
```

```
2256                }
2257                vec += btopr(ssize);
2258        }
2259        AS_LOCK_EXIT(as);
2259        AS_LOCK_EXIT(as, &as->a_lock);
2260        return (error);
2261 }
_____unchanged_portion_omitted_

2307 /*
2308  * Cache control operations over the interval [addr, addr + size) in
2309  * address space "as".
2310  */
2311 /*ARGSUSED*/
2312 int
2313 as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
2314     uintptr_t arg, ulong_t *lock_map, size_t pos)
2315 {
2316        struct seg *seg;        /* working segment */
2317        caddr_t raddr;          /* rounded down addr */
2318        caddr_t initraddr;      /* saved initial rounded down addr */
2319        size_t rsize;           /* rounded up size */
2320        size_t initrsize;       /* saved initial rounded up size */
2321        size_t ssize;           /* size of seg */
2322        int error = 0;                  /* result */
2323        size_t mlock_size;      /* size of bitmap */
2324        ulong_t *mlock_map;     /* pointer to bitmap used */
2325                                /* to represent the locked */
2326                                /* pages. */
2327 retry:
2328        if (error == IE_RETRY)
2329                AS_LOCK_ENTER(as, RW_WRITER);
2329                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2330        else
2331                AS_LOCK_ENTER(as, RW_READER);
2331                AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2333        /*
2334         * If these are address space lock/unlock operations, loop over
2335         * all segments in the address space, as appropriate.
2336         */
2337        if (func == MC_LOCKAS) {
2338                size_t npages, idx;
2339                size_t rlen = 0;        /* rounded as length */

2341                idx = pos;

2343                if (arg & MCL_FUTURE) {
2344                        mutex_enter(&as->a_contents);
2345                        AS_SETPGLCK(as);
2346                        mutex_exit(&as->a_contents);
2347                }
2348                if ((arg & MCL_CURRENT) == 0) {
2349                        AS_LOCK_EXIT(as);
2349                        AS_LOCK_EXIT(as, &as->a_lock);
2350                        return (0);
2351                }

2353                seg = AS_SEGFIRST(as);
2354                if (seg == NULL) {
2355                        AS_LOCK_EXIT(as);
2355                        AS_LOCK_EXIT(as, &as->a_lock);
2356                        return (0);
2357                }

2359                do {
```



```
2360                        raddr = (caddr_t)((uintptr_t)seg->s_base &
2361                            (uintptr_t)PAGEMASK);
2362                        rlen += (((uintptr_t)(seg->s_base + seg->s_size) +
2363                            PAGEOFFSET) & PAGEMASK) - (uintptr_t)raddr;
2364                } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2366                mlock_size = BT_BITOUL(btopr(rlen));
2367                if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2368                    sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2369                                AS_LOCK_EXIT(as);
2369                                AS_LOCK_EXIT(as, &as->a_lock);
2370                                return (EAGAIN);
2371                }

2373                for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2374                        error = SEGOP_LOCKOP(seg, seg->s_base,
2375                            seg->s_size, attr, MC_LOCK, mlock_map, pos);
2376                        if (error != 0)
2377                                break;
2378                        pos += seg_pages(seg);
2379                }

2381                if (error) {
2382                        for (seg = AS_SEGFIRST(as); seg != NULL;
2383                            seg = AS_SEGNEXT(as, seg)) {

2385                                raddr = (caddr_t)((uintptr_t)seg->s_base &
2386                                    (uintptr_t)PAGEMASK);
2387                                npages = seg_pages(seg);
2388                                as_segunlock(seg, raddr, attr, mlock_map,
2389                                    idx, npages);
2390                                idx += npages;
2391                        }
2392                }

2394                kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2395                AS_LOCK_EXIT(as);
2395                AS_LOCK_EXIT(as, &as->a_lock);
2396                goto lockerr;
2397        } else if (func == MC_UNLOCKAS) {
2398                mutex_enter(&as->a_contents);
2399                AS_CLRPGLCK(as);
2400                mutex_exit(&as->a_contents);

2402                for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2403                        error = SEGOP_LOCKOP(seg, seg->s_base,
2404                            seg->s_size, attr, MC_UNLOCK, NULL, 0);
2405                        if (error != 0)
2406                                break;
2407                }

2409                AS_LOCK_EXIT(as);
2409                AS_LOCK_EXIT(as, &as->a_lock);
2410                goto lockerr;
2411        }

2413        /*
2414         * Normalize addresses and sizes.
2415         */
2416        initraddr = raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2417        initrsize = rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2418            (size_t)raddr;

2420        if (raddr + rsize < raddr) {            /* check for wraparound */
2421                AS_LOCK_EXIT(as);
2421                AS_LOCK_EXIT(as, &as->a_lock);
```

```
2422                    return (ENOMEM);
2423            }

2425            /*
2426             * Get initial segment.
2427             */
2428            if ((seg = as_segat(as, raddr)) == NULL) {
2429                    AS_LOCK_EXIT(as);
2429                    AS_LOCK_EXIT(as, &as->a_lock);
2430                    return (ENOMEM);
2431            }

2433            if (func == MC_LOCK) {
2434                    mlock_size = BT_BITOUL(btopr(rsize));
2435                    if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2436                        sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2437                                    AS_LOCK_EXIT(as);
2437                                    AS_LOCK_EXIT(as, &as->a_lock);
2438                                    return (EAGAIN);
2439                    }
2440            }

2442            /*
2443             * Loop over all segments.  If a hole in the address range is
2444             * discovered, then fail.  For each segment, perform the appropriate
2445             * control operation.
2446             */
2447            while (rsize != 0) {

2449                    /*
2450                     * Make sure there's no hole, calculate the portion
2451                     * of the next segment to be operated over.
2452                     */
2453                    if (raddr >= seg->s_base + seg->s_size) {
2454                            seg = AS_SEGNEXT(as, seg);
2455                            if (seg == NULL || raddr != seg->s_base) {
2456                                    if (func == MC_LOCK) {
2457                                            as_unlockerr(as, attr, mlock_map,
2458                                                initraddr, initrsize - rsize);
2459                                            kmem_free(mlock_map,
2460                                                mlock_size * sizeof (ulong_t));
2461                                    }
2462                                    AS_LOCK_EXIT(as);
2462                                    AS_LOCK_EXIT(as, &as->a_lock);
2463                                    return (ENOMEM);
2464                            }
2465                    }
2466                    if ((raddr + rsize) > (seg->s_base + seg->s_size))
2467                            ssize = seg->s_base + seg->s_size - raddr;
2468                    else
2469                            ssize = rsize;

2471                    /*
2472                     * Dispatch on specific function.
2473                     */
2474                    switch (func) {

2476                    /*
2477                     * Synchronize cached data from mappings with backing
2478                     * objects.
2479                     */
2480                    case MC_SYNC:
2481                            if (error = SEGOP_SYNC(seg, raddr, ssize,
2482                                attr, (uint_t)arg)) {
2483                                    AS_LOCK_EXIT(as);
2483                                    AS_LOCK_EXIT(as, &as->a_lock);
```



```
2484                                    return (error);
2485                            }
2486                            break;

2488                    /*
2489                     * Lock pages in memory.
2490                     */
2491                    case MC_LOCK:
2492                            if (error = SEGOP_LOCKOP(seg, raddr, ssize,
2493                                attr, func, mlock_map, pos)) {
2494                                    as_unlockerr(as, attr, mlock_map, initraddr,
2495                                        initrsize - rsize + ssize);
2496                                    kmem_free(mlock_map, mlock_size *
2497                                        sizeof (ulong_t));
2498                                    AS_LOCK_EXIT(as);
2498                                    AS_LOCK_EXIT(as, &as->a_lock);
2499                                    goto lockerr;
2500                            }
2501                            break;

2503                    /*
2504                     * Unlock mapped pages.
2505                     */
2506                    case MC_UNLOCK:
2507                            (void) SEGOP_LOCKOP(seg, raddr, ssize, attr, func,
2508                                (ulong_t *)NULL, (size_t)NULL);
2509                            break;

2511                    /*
2512                     * Store VM advise for mapped pages in segment layer.
2513                     */
2514                    case MC_ADVISE:
2515                            error = SEGOP_ADVISE(seg, raddr, ssize, (uint_t)arg);

2517                            /*
2518                             * Check for regular errors and special retry error
2519                             */
2520                            if (error) {
2521                                    if (error == IE_RETRY) {
2522                                            /*
2523                                             * Need to acquire writers lock, so
2524                                             * have to drop readers lock and start
2525                                             * all over again
2526                                             */
2527                                            AS_LOCK_EXIT(as);
2527                                            AS_LOCK_EXIT(as, &as->a_lock);
2528                                            goto retry;
2529                                    } else if (error == IE_REATTACH) {
2530                                            /*
2531                                             * Find segment for current address
2532                                             * because current segment just got
2533                                             * split or concatenated
2534                                             */
2535                                            seg = as_segat(as, raddr);
2536                                            if (seg == NULL) {
2537                                                    AS_LOCK_EXIT(as);
2537                                                    AS_LOCK_EXIT(as, &as->a_lock);
2538                                                    return (ENOMEM);
2539                                            }
2540                                    } else {
2541                                            /*
2542                                             * Regular error
2543                                             */
2544                                            AS_LOCK_EXIT(as);
2544                                            AS_LOCK_EXIT(as, &as->a_lock);
2545                                            return (error);
```

```
2546                                    }
2547                            }
2548                            break;

2550                    case MC_INHERIT_ZERO:
2551                            if (seg->s_ops->inherit == NULL) {
2552                                    error = ENOTSUP;
2553                            } else {
2554                                    error = SEGOP_INHERIT(seg, raddr, ssize,
2555                                        SEGP_INH_ZERO);
2556                            }
2557                            if (error != 0) {
2558                                    AS_LOCK_EXIT(as);
2558                                    AS_LOCK_EXIT(as, &as->a_lock);
2559                                    return (error);
2560                            }
2561                            break;

2563                    /*
2564                     * Can't happen.
2565                     */
2566                    default:
2567                            panic("as_ctl: bad operation %d", func);
2568                            /*NOTREACHED*/
2569                    }

2571                    rsize -= ssize;
2572                    raddr += ssize;
2573            }

2575            if (func == MC_LOCK)
2576                    kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2577            AS_LOCK_EXIT(as);
2577            AS_LOCK_EXIT(as, &as->a_lock);
2578            return (0);
2579    lockerr:

2581            /*
2582             * If the lower levels returned EDEADLK for a segment lockop,
2583             * it means that we should retry the operation.  Let's wait
2584             * a bit also to let the deadlock causing condition clear.
2585             * This is part of a gross hack to work around a design flaw
2586             * in the ufs/sds logging code and should go away when the
2587             * logging code is re-designed to fix the problem. See bug
2588             * 4125102 for details of the problem.
2589             */
2590            if (error == EDEADLK) {
2591                    delay(deadlk_wait);
2592                    error = 0;
2593                    goto retry;
2594            }
2595            return (error);
2596    }
______unchanged_portion_omitted_

2617 /*
2618  * Pagelock pages from a range that spans more than 1 segment.  Obtain shadow
2619  * lists from each segment and copy them to one contiguous shadow list (plist)
2620  * as expected by the caller.  Save pointers to per segment shadow lists at
2621  * the tail of plist so that they can be used during as_pageunlock().
2622  */
2623 static int
2624 as_pagelock_segs(struct as *as, struct seg *seg, struct page ***ppp,
2625     caddr_t addr, size_t size, enum seg_rw rw)
2626 {
2627            caddr_t sv_addr = addr;
```



```
2628            size_t sv_size = size;
2629            struct seg *sv_seg = seg;
2630            ulong_t segcnt = 1;
2631            ulong_t cnt;
2632            size_t ssize;
2633            pgcnt_t npages = btop(size);
2634            page_t **plist;
2635            page_t **pl;
2636            int error;
2637            caddr_t eaddr;
2638            faultcode_t fault_err = 0;
2639            pgcnt_t pl_off;
2640            extern struct seg_ops segspt_shmops;

2642            ASSERT(AS_LOCK_HELD(as));
2642            ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2643            ASSERT(seg != NULL);
2644            ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2645            ASSERT(addr + size > seg->s_base + seg->s_size);
2646            ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2647            ASSERT(IS_P2ALIGNED(addr, PAGESIZE));

2649            /*
2650             * Count the number of segments covered by the range we are about to
2651             * lock. The segment count is used to size the shadow list we return
2652             * back to the caller.
2653             */
2654            for (; size != 0; size -= ssize, addr += ssize) {
2655                    if (addr >= seg->s_base + seg->s_size) {

2657                            seg = AS_SEGNEXT(as, seg);
2658                            if (seg == NULL || addr != seg->s_base) {
2659                                    AS_LOCK_EXIT(as);
2659                                    AS_LOCK_EXIT(as, &as->a_lock);
2660                                    return (EFAULT);
2661                            }
2662                            /*
2663                             * Do a quick check if subsequent segments
2664                             * will most likely support pagelock.
2665                             */
2666                            if (seg->s_ops == &segvn_ops) {
2667                                    vnode_t *vp;

2669                                    if (SEGOP_GETVP(seg, addr, &vp) != 0 ||
2670                                        vp != NULL) {
2671                                            AS_LOCK_EXIT(as);
2671                                            AS_LOCK_EXIT(as, &as->a_lock);
2672                                            goto slow;
2673                                    }
2674                            } else if (seg->s_ops != &segspt_shmops) {
2675                                    AS_LOCK_EXIT(as);
2675                                    AS_LOCK_EXIT(as, &as->a_lock);
2676                                    goto slow;
2677                            }
2678                            segcnt++;
2679                    }
2680                    if (addr + size > seg->s_base + seg->s_size) {
2681                            ssize = seg->s_base + seg->s_size - addr;
2682                    } else {
2683                            ssize = size;
2684                    }
2685            }
2686            ASSERT(segcnt > 1);

2688            plist = kmem_zalloc((npages + segcnt) * sizeof (page_t *), KM_SLEEP);
```

```
2690         addr = sv_addr;
2691         size = sv_size;
2692         seg = sv_seg;

2694         for (cnt = 0, pl_off = 0; size != 0; size -= ssize, addr += ssize) {
2695                 if (addr >= seg->s_base + seg->s_size) {
2696                         seg = AS_SEGNEXT(as, seg);
2697                         ASSERT(seg != NULL && addr == seg->s_base);
2698                         cnt++;
2699                         ASSERT(cnt < segcnt);
2700                 }
2701                 if (addr + size > seg->s_base + seg->s_size) {
2702                         ssize = seg->s_base + seg->s_size - addr;
2703                 } else {
2704                         ssize = size;
2705                 }
2706                 pl = &plist[npages + cnt];
2707                 error = SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2708                     L_PAGELOCK, rw);
2709                 if (error) {
2710                         break;
2711                 }
2712                 ASSERT(plist[npages + cnt] != NULL);
2713                 ASSERT(pl_off + btop(ssize) <= npages);
2714                 bcopy(plist[npages + cnt], &plist[pl_off],
2715                     btop(ssize) * sizeof (page_t *));
2716                 pl_off += btop(ssize);
2717         }

2719         if (size == 0) {
2720                 AS_LOCK_EXIT(as);
2720                 AS_LOCK_EXIT(as, &as->a_lock);
2721                 ASSERT(cnt == segcnt - 1);
2722                 *ppp = plist;
2723                 return (0);
2724         }

2726         /*
2727          * one of pagelock calls failed. The error type is in error variable.
2728          * Unlock what we've locked so far and retry with F_SOFTLOCK if error
2729          * type is either EFAULT or ENOTSUP. Otherwise just return the error
2730          * back to the caller.
2731          */

2733         eaddr = addr;
2734         seg = sv_seg;

2736         for (cnt = 0, addr = sv_addr; addr < eaddr; addr += ssize) {
2737                 if (addr >= seg->s_base + seg->s_size) {
2738                         seg = AS_SEGNEXT(as, seg);
2739                         ASSERT(seg != NULL && addr == seg->s_base);
2740                         cnt++;
2741                         ASSERT(cnt < segcnt);
2742                 }
2743                 if (eaddr > seg->s_base + seg->s_size) {
2744                         ssize = seg->s_base + seg->s_size - addr;
2745                 } else {
2746                         ssize = eaddr - addr;
2747                 }
2748                 pl = &plist[npages + cnt];
2749                 ASSERT(*pl != NULL);
2750                 (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2751                     L_PAGEUNLOCK, rw);
2752         }

2754         AS_LOCK_EXIT(as);
```



```
2754         AS_LOCK_EXIT(as, &as->a_lock);

2756         kmem_free(plist, (npages + segcnt) * sizeof (page_t *));

2758         if (error != ENOTSUP && error != EFAULT) {
2759                 return (error);
2760         }

2762 slow:
2763         /*
2764          * If we are here because pagelock failed due to the need to cow fault
2765          * in the pages we want to lock F_SOFTLOCK will do this job and in
2766          * next as_pagelock() call for this address range pagelock will
2767          * hopefully succeed.
2768          */
2769         fault_err = as_fault(as->a_hat, as, sv_addr, sv_size, F_SOFTLOCK, rw);
2770         if (fault_err != 0) {
2771                 return (fc_decode(fault_err));
2772         }
2773         *ppp = NULL;

2775         return (0);
2776 }

2778 /*
2779  * lock pages in a given address space. Return shadow list. If
2780  * the list is NULL, the MMU mapping is also locked.
2781  */
2782 int
2783 as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
2784     size_t size, enum seg_rw rw)
2785 {
2786         size_t rsize;
2787         caddr_t raddr;
2788         faultcode_t fault_err;
2789         struct seg *seg;
2790         int err;

2792         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_START,
2793             "as_pagelock_start: addr %p size %ld", addr, size);

2795         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2796         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2797             (size_t)raddr;

2799         /*
2800          * if the request crosses two segments let
2801          * as_fault handle it.
2802          */
2803         AS_LOCK_ENTER(as, RW_READER);
2803         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2805         seg = as_segat(as, raddr);
2806         if (seg == NULL) {
2807                 AS_LOCK_EXIT(as);
2807                 AS_LOCK_EXIT(as, &as->a_lock);
2808                 return (EFAULT);
2809         }
2810         ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2811         if (raddr + rsize > seg->s_base + seg->s_size) {
2812                 return (as_pagelock_segs(as, seg, ppp, raddr, rsize, rw));
2813         }
2814         if (raddr + rsize <= raddr) {
2815                 AS_LOCK_EXIT(as);
2815                 AS_LOCK_EXIT(as, &as->a_lock);
2816                 return (EFAULT);
```

```
2817         }

2819         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_START,
2820             "seg_lock_1_start: raddr %p rsize %ld", raddr, rsize);

2822         /*
2823          * try to lock pages and pass back shadow list
2824          */
2825         err = SEGOP_PAGELOCK(seg, raddr, rsize, ppp, L_PAGELOCK, rw);

2827         TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_END, "seg_lock_1_end");

2829         AS_LOCK_EXIT(as);
2829         AS_LOCK_EXIT(as, &as->a_lock);

2831         if (err == 0 || (err != ENOTSUP && err != EFAULT)) {
2832                 return (err);
2833         }

2835         /*
2836          * Use F_SOFTLOCK to lock the pages because pagelock failed either due
2837          * to no pagelock support for this segment or pages need to be cow
2838          * faulted in. If fault is needed F_SOFTLOCK will do this job for
2839          * this as_pagelock() call and in the next as_pagelock() call for the
2840          * same address range pagelock call will hopefull succeed.
2841          */
2842         fault_err = as_fault(as->a_hat, as, addr, size, F_SOFTLOCK, rw);
2843         if (fault_err != 0) {
2844                 return (fc_decode(fault_err));
2845         }
2846         *ppp = NULL;

2848         TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_END, "as_pagelock_end");
2849         return (0);
2850 }

2852 /*
2853  * unlock pages locked by as_pagelock_segs().  Retrieve per segment shadow
2854  * lists from the end of plist and call pageunlock interface for each segment.
2855  * Drop as lock and free plist.
2856  */
2857 static void
2858 as_pageunlock_segs(struct as *as, struct seg *seg, caddr_t addr, size_t size,
2859     struct page **plist, enum seg_rw rw)
2860 {
2861         ulong_t cnt;
2862         caddr_t eaddr = addr + size;
2863         pgcnt_t npages = btop(size);
2864         size_t ssize;
2865         page_t **pl;

2867         ASSERT(AS_LOCK_HELD(as));
2867         ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2868         ASSERT(seg != NULL);
2869         ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2870         ASSERT(addr + size > seg->s_base + seg->s_size);
2871         ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2872         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
2873         ASSERT(plist != NULL);

2875         for (cnt = 0; addr < eaddr; addr += ssize) {
2876                 if (addr >= seg->s_base + seg->s_size) {
2877                         seg = AS_SEGNEXT(as, seg);
2878                         ASSERT(seg != NULL && addr == seg->s_base);
2879                         cnt++;
2880                 }
```



```
2881                 if (eaddr > seg->s_base + seg->s_size) {
2882                         ssize = seg->s_base + seg->s_size - addr;
2883                 } else {
2884                         ssize = eaddr - addr;
2885                 }
2886                 pl = &plist[npages + cnt];
2887                 ASSERT(*pl != NULL);
2888                 (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2889                     L_PAGEUNLOCK, rw);
2890         }
2891         ASSERT(cnt > 0);
2892         AS_LOCK_EXIT(as);
2892         AS_LOCK_EXIT(as, &as->a_lock);

2894         cnt++;
2895         kmem_free(plist, (npages + cnt) * sizeof (page_t *));
2896 }

2898 /*
2899  * unlock pages in a given address range
2900  */
2901 void
2902 as_pageunlock(struct as *as, struct page **pp, caddr_t addr, size_t size,
2903     enum seg_rw rw)
2904 {
2905         struct seg *seg;
2906         size_t rsize;
2907         caddr_t raddr;

2909         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_START,
2910             "as_pageunlock_start: addr %p size %ld", addr, size);

2912         /*
2913          * if the shadow list is NULL, as_pagelock was
2914          * falling back to as_fault
2915          */
2916         if (pp == NULL) {
2917                 (void) as_fault(as->a_hat, as, addr, size, F_SOFTUNLOCK, rw);
2918                 return;
2919         }

2921         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2922         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2923             (size_t)raddr;

2925         AS_LOCK_ENTER(as, RW_READER);
2925         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2926         seg = as_segat(as, raddr);
2927         ASSERT(seg != NULL);

2929         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_UNLOCK_START,
2930             "seg_unlock_start: raddr %p rsize %ld", raddr, rsize);

2932         ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2933         if (raddr + rsize <= seg->s_base + seg->s_size) {
2934                 SEGOP_PAGELOCK(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2935         } else {
2936                 as_pageunlock_segs(as, seg, raddr, rsize, pp, rw);
2937                 return;
2938         }
2939         AS_LOCK_EXIT(as);
2939         AS_LOCK_EXIT(as, &as->a_lock);
2940         TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_END, "as_pageunlock_end");
2941 }

2943 int
```

```
2944 as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
2945     boolean_t wait)
2946 {
2947         struct seg *seg;
2948         size_t ssize;
2949         caddr_t raddr;                  /* rounded down addr */
2950         size_t rsize;                   /* rounded up size */
2951         int error = 0;
2952         size_t pgsz = page_get_pagesize(szc);

2954 setpgsz_top:
2955         if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(size, pgsz)) {
2956                 return (EINVAL);
2957         }

2959         raddr = addr;
2960         rsize = size;

2962         if (raddr + rsize < raddr)              /* check for wraparound */
2963                 return (ENOMEM);

2965         AS_LOCK_ENTER(as, RW_WRITER);
2965         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2966         as_clearwatchprot(as, raddr, rsize);
2967         seg = as_segat(as, raddr);
2968         if (seg == NULL) {
2969                 as_setwatch(as);
2970                 AS_LOCK_EXIT(as);
2970                 AS_LOCK_EXIT(as, &as->a_lock);
2971                 return (ENOMEM);
2972         }

2974         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2975                 if (raddr >= seg->s_base + seg->s_size) {
2976                         seg = AS_SEGNEXT(as, seg);
2977                         if (seg == NULL || raddr != seg->s_base) {
2978                                 error = ENOMEM;
2979                                 break;
2980                         }
2981                 }
2982                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2983                         ssize = seg->s_base + seg->s_size - raddr;
2984                 } else {
2985                         ssize = rsize;
2986                 }

2988 retry:
2989                 error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);

2991                 if (error == IE_NOMEM) {
2992                         error = EAGAIN;
2993                         break;
2994                 }

2996                 if (error == IE_RETRY) {
2997                         AS_LOCK_EXIT(as);
2997                         AS_LOCK_EXIT(as, &as->a_lock);
2998                         goto setpgsz_top;
2999                 }

3001                 if (error == ENOTSUP) {
3002                         error = EINVAL;
3003                         break;
3004                 }

3006                 if (wait && (error == EAGAIN)) {
```



```
3007                         /*
3008                          * Memory is currently locked.  It must be unlocked
3009                          * before this operation can succeed through a retry.
3010                          * The possible reasons for locked memory and
3011                          * corresponding strategies for unlocking are:
3012                          * (1) Normal I/O
3013                          *      wait for a signal that the I/O operation
3014                          *      has completed and the memory is unlocked.
3015                          * (2) Asynchronous I/O
3016                          *      The aio subsystem does not unlock pages when
3017                          *      the I/O is completed. Those pages are unlocked
3018                          *      when the application calls aiowait/aioerror.
3019                          *      So, to prevent blocking forever, cv_broadcast()
3020                          *      is done to wake up aio_cleanup_thread.
3021                          *      Subsequently, segvn_reclaim will be called, and
3022                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
3023                          * (3) Long term page locking:
3024                          *      This is not relevant for as_setpagesize()
3025                          *      because we cannot change the page size for
3026                          *      driver memory. The attempt to do so will
3027                          *      fail with a different error than EAGAIN so
3028                          *      there's no need to trigger as callbacks like
3029                          *      as_unmap, as_setprot or as_free would do.
3030                          */
3031                         mutex_enter(&as->a_contents);
3032                         if (!AS_ISNOUNMAPWAIT(as)) {
3033                                 if (AS_ISUNMAPWAIT(as) == 0) {
3034                                         cv_broadcast(&as->a_cv);
3035                                 }
3036                                 AS_SETUNMAPWAIT(as);
3037                                 AS_LOCK_EXIT(as);
3037                                 AS_LOCK_EXIT(as, &as->a_lock);
3038                                 while (AS_ISUNMAPWAIT(as)) {
3039                                         cv_wait(&as->a_cv, &as->a_contents);
3040                                 }
3041                         } else {
3042                                 /*
3043                                  * We may have raced with
3044                                  * segvn_reclaim()/segspt_reclaim(). In this
3045                                  * case clean nounmapwait flag and retry since
3046                                  * softlockcnt in this segment may be already
3047                                  * 0.  We don't drop as writer lock so our
3048                                  * number of retries without sleeping should
3049                                  * be very small. See segvn_reclaim() for
3050                                  * more comments.
3051                                  */
3052                                 AS_CLRNOUNMAPWAIT(as);
3053                                 mutex_exit(&as->a_contents);
3054                                 goto retry;
3055                         }
3056                         mutex_exit(&as->a_contents);
3057                         goto setpgsz_top;
3058                 } else if (error != 0) {
3059                         break;
3060                 }
3061         }
3062         as_setwatch(as);
3063         AS_LOCK_EXIT(as);
3063         AS_LOCK_EXIT(as, &as->a_lock);
3064         return (error);
3065 }

3067 /*
3068  * as_iset3_default_lpsize() just calls SEGOP_SETPAGESIZE() on all segments
3069  * in its chunk where s_szc is less than the szc we want to set.
3070  */
```

```
3071 static int
3072 as_iset3_default_lpsize(struct as *as, caddr_t raddr, size_t rsize, uint_t szc,
3073     int *retry)
3074 {
3075         struct seg *seg;
3076         size_t ssize;
3077         int error;

3079         ASSERT(AS_WRITE_HELD(as));
3079         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3081         seg = as_segat(as, raddr);
3082         if (seg == NULL) {
3083                 panic("as_iset3_default_lpsize: no seg");
3084         }

3086         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
3087                 if (raddr >= seg->s_base + seg->s_size) {
3088                         seg = AS_SEGNEXT(as, seg);
3089                         if (seg == NULL || raddr != seg->s_base) {
3090                                 panic("as_iset3_default_lpsize: as changed");
3091                         }
3092                 }
3093                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3094                         ssize = seg->s_base + seg->s_size - raddr;
3095                 } else {
3096                         ssize = rsize;
3097                 }

3099                 if (szc > seg->s_szc) {
3100                         error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);
3101                         /* Only retry on EINVAL segments that have no vnode. */
3102                         if (error == EINVAL) {
3103                                 vnode_t *vp = NULL;
3104                                 if ((SEGOP_GETTYPE(seg, raddr) & MAP_SHARED) &&
3105                                     (SEGOP_GETVP(seg, raddr, &vp) != 0 ||
3106                                     vp == NULL)) {
3107                                         *retry = 1;
3108                                 } else {
3109                                         *retry = 0;
3110                                 }
3111                         }
3112                         if (error) {
3113                                 return (error);
3114                         }
3115                 }
3116         }
3117         return (0);
3118 }

3120 /*
3121  * as_iset2_default_lpsize() calls as_iset3_default_lpsize() to set the
3122  * pagesize on each segment in its range, but if any fails with EINVAL,
3123  * then it reduces the pagesizes to the next size in the bitmap and
3124  * retries as_iset3_default_lpsize(). The reason why the code retries
3125  * smaller allowed sizes on EINVAL is because (a) the anon offset may not
3126  * match the bigger sizes, and (b) it's hard to get this offset (to begin
3127  * with) to pass to map_pgszcvec().
3128  */
3129 static int
3130 as_iset2_default_lpsize(struct as *as, caddr_t addr, size_t size, uint_t szc,
3131     uint_t szcvec)
3132 {
3133         int error;
3134         int retry;
```



```
3136         ASSERT(AS_WRITE_HELD(as));
3136         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3138         for (;;) {
3139                 error = as_iset3_default_lpsize(as, addr, size, szc, &retry);
3140                 if (error == EINVAL && retry) {
3141                         szcvec &= ~(1 << szc);
3142                         if (szcvec <= 1) {
3143                                 return (EINVAL);
3144                         }
3145                         szc = highbit(szcvec) - 1;
3146                 } else {
3147                         return (error);
3148                 }
3149         }
3150 }

3152 /*
3153  * as_iset1_default_lpsize() breaks its chunk into areas where existing
3154  * segments have a smaller szc than we want to set. For each such area,
3155  * it calls as_iset2_default_lpsize()
3156  */
3157 static int
3158 as_iset1_default_lpsize(struct as *as, caddr_t raddr, size_t rsize, uint_t szc,
3159     uint_t szcvec)
3160 {
3161         struct seg *seg;
3162         size_t ssize;
3163         caddr_t setaddr = raddr;
3164         size_t setsize = 0;
3165         int set;
3166         int error;

3168         ASSERT(AS_WRITE_HELD(as));
3168         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3170         seg = as_segat(as, raddr);
3171         if (seg == NULL) {
3172                 panic("as_iset1_default_lpsize: no seg");
3173         }
3174         if (seg->s_szc < szc) {
3175                 set = 1;
3176         } else {
3177                 set = 0;
3178         }

3180         for (; rsize != 0; rsize -= ssize, raddr += ssize, setsize += ssize) {
3181                 if (raddr >= seg->s_base + seg->s_size) {
3182                         seg = AS_SEGNEXT(as, seg);
3183                         if (seg == NULL || raddr != seg->s_base) {
3184                                 panic("as_iset1_default_lpsize: as changed");
3185                         }
3186                         if (seg->s_szc >= szc && set) {
3187                                 ASSERT(setsize != 0);
3188                                 error = as_iset2_default_lpsize(as,
3189                                     setaddr, setsize, szc, szcvec);
3190                                 if (error) {
3191                                         return (error);
3192                                 }
3193                                 set = 0;
3194                         } else if (seg->s_szc < szc && !set) {
3195                                 setaddr = raddr;
3196                                 setsize = 0;
3197                                 set = 1;
3198                         }
3199                 }
```

```
3200                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3201                         ssize = seg->s_base + seg->s_size - raddr;
3202                 } else {
3203                         ssize = rsize;
3204                 }
3205         }
3206         error = 0;
3207         if (set) {
3208                 ASSERT(setsize != 0);
3209                 error = as_iset2_default_lpsize(as, setaddr, setsize,
3210                     szc, szcvec);
3211         }
3212         return (error);
3213 }

3215 /*
3216  * as_iset_default_lpsize() breaks its chunk according to the size code bitmap
3217  * returned by map_pgszcvec() (similar to as_map_segvn_segs()), and passes each
3218  * chunk to as_iset1_default_lpsize().
3219  */
3220 static int
3221 as_iset_default_lpsize(struct as *as, caddr_t addr, size_t size, int flags,
3222     int type)
3223 {
3224         int rtype = (type & MAP_SHARED) ? MAPPGSZC_SHM : MAPPGSZC_PRIVM;
3225         uint_t szcvec = map_pgszcvec(addr, size, (uintptr_t)addr,
3226             flags, rtype, 1);
3227         uint_t szc;
3228         uint_t nszc;
3229         int error;
3230         caddr_t a;
3231         caddr_t eaddr;
3232         size_t segsize;
3233         size_t pgsz;
3234         uint_t save_szcvec;

3236         ASSERT(AS_WRITE_HELD(as));
3236         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
3237         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
3238         ASSERT(IS_P2ALIGNED(size, PAGESIZE));

3240         szcvec &= ~1;
3241         if (szcvec <= 1) {      /* skip if base page size */
3242                 return (0);
3243         }

3245         /* Get the pagesize of the first larger page size. */
3246         szc = lowbit(szcvec) - 1;
3247         pgsz = page_get_pagesize(szc);
3248         eaddr = addr + size;
3249         addr = (caddr_t)P2ROUNDUP((uintptr_t)addr, pgsz);
3250         eaddr = (caddr_t)P2ALIGN((uintptr_t)eaddr, pgsz);

3252         save_szcvec = szcvec;
3253         szcvec >>= (szc + 1);
3254         nszc = szc;
3255         while (szcvec) {
3256                 if ((szcvec & 0x1) == 0) {
3257                         nszc++;
3258                         szcvec >>= 1;
3259                         continue;
3260                 }
3261                 nszc++;
3262                 pgsz = page_get_pagesize(nszc);
3263                 a = (caddr_t)P2ROUNDUP((uintptr_t)addr, pgsz);
3264                 if (a != addr) {
```



```
3265                         ASSERT(szc > 0);
3266                         ASSERT(a < eaddr);
3267                         segsize = a - addr;
3268                         error = as_iset1_default_lpsize(as, addr, segsize, szc,
3269                             save_szcvec);
3270                         if (error) {
3271                                 return (error);
3272                         }
3273                         addr = a;
3274                 }
3275                 szc = nszc;
3276                 szcvec >>= 1;
3277         }

3279         ASSERT(addr < eaddr);
3280         szcvec = save_szcvec;
3281         while (szcvec) {
3282                 a = (caddr_t)P2ALIGN((uintptr_t)eaddr, pgsz);
3283                 ASSERT(a >= addr);
3284                 if (a != addr) {
3285                         ASSERT(szc > 0);
3286                         segsize = a - addr;
3287                         error = as_iset1_default_lpsize(as, addr, segsize, szc,
3288                             save_szcvec);
3289                         if (error) {
3290                                 return (error);
3291                         }
3292                         addr = a;
3293                 }
3294                 szcvec &= ~(1 << szc);
3295                 if (szcvec) {
3296                         szc = highbit(szcvec) - 1;
3297                         pgsz = page_get_pagesize(szc);
3298                 }
3299         }
3300         ASSERT(addr == eaddr);

3302         return (0);
3303 }

3305 /*
3306  * Set the default large page size for the range. Called via memcntl with
3307  * page size set to 0. as_set_default_lpsize breaks the range down into
3308  * chunks with the same type/flags, ignores-non segvn segments, and passes
3309  * each chunk to as_iset_default_lpsize().
3310  */
3311 int
3312 as_set_default_lpsize(struct as *as, caddr_t addr, size_t size)
3313 {
3314         struct seg *seg;
3315         caddr_t raddr;
3316         size_t rsize;
3317         size_t ssize;
3318         int rtype, rflags;
3319         int stype, sflags;
3320         int error;
3321         caddr_t setaddr;
3322         size_t setsize;
3323         int segvn;

3325         if (size == 0)
3326                 return (0);

3328         AS_LOCK_ENTER(as, RW_WRITER);
3328         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3329 again:
```

```
3330         error = 0;

3332         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3333         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
3334             (size_t)raddr;

3336         if (raddr + rsize < raddr) {            /* check for wraparound */
3337                 AS_LOCK_EXIT(as);
3337                 AS_LOCK_EXIT(as, &as->a_lock);
3338                 return (ENOMEM);
3339         }
3340         as_clearwatchprot(as, raddr, rsize);
3341         seg = as_segat(as, raddr);
3342         if (seg == NULL) {
3343                 as_setwatch(as);
3344                 AS_LOCK_EXIT(as);
3344                 AS_LOCK_EXIT(as, &as->a_lock);
3345                 return (ENOMEM);
3346         }
3347         if (seg->s_ops == &segvn_ops) {
3348                 rtype = SEGOP_GETTYPE(seg, addr);
3349                 rflags = rtype & (MAP_TEXT | MAP_INITDATA);
3350                 rtype = rtype & (MAP_SHARED | MAP_PRIVATE);
3351                 segvn = 1;
3352         } else {
3353                 segvn = 0;
3354         }
3355         setaddr = raddr;
3356         setsize = 0;

3358         for (; rsize != 0; rsize -= ssize, raddr += ssize, setsize += ssize) {
3359                 if (raddr >= (seg->s_base + seg->s_size)) {
3360                         seg = AS_SEGNEXT(as, seg);
3361                         if (seg == NULL || raddr != seg->s_base) {
3362                                 error = ENOMEM;
3363                                 break;
3364                         }
3365                         if (seg->s_ops == &segvn_ops) {
3366                                 stype = SEGOP_GETTYPE(seg, raddr);
3367                                 sflags = stype & (MAP_TEXT | MAP_INITDATA);
3368                                 stype &= (MAP_SHARED | MAP_PRIVATE);
3369                                 if (segvn && (rflags != sflags ||
3370                                     rtype != stype)) {
3371                                         /*
3372                                          * The next segment is also segvn but
3373                                          * has different flags and/or type.
3374                                          */
3375                                         ASSERT(setsize != 0);
3376                                         error = as_iset_default_lpsize(as,
3377                                             setaddr, setsize, rflags, rtype);
3378                                         if (error) {
3379                                                 break;
3380                                         }
3381                                         rflags = sflags;
3382                                         rtype = stype;
3383                                         setaddr = raddr;
3384                                         setsize = 0;
3385                                 } else if (!segvn) {
3386                                         rflags = sflags;
3387                                         rtype = stype;
3388                                         setaddr = raddr;
3389                                         setsize = 0;
3390                                         segvn = 1;
3391                                 }
3392                         } else if (segvn) {
3393                                 /* The next segment is not segvn. */
```



```
3394                                 ASSERT(setsize != 0);
3395                                 error = as_iset_default_lpsize(as,
3396                                     setaddr, setsize, rflags, rtype);
3397                                 if (error) {
3398                                         break;
3399                                 }
3400                                 segvn = 0;
3401                         }
3402                 }
3403                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3404                         ssize = seg->s_base + seg->s_size - raddr;
3405                 } else {
3406                         ssize = rsize;
3407                 }
3408         }
3409         if (error == 0 && segvn) {
3410                 /* The last chunk when rsize == 0. */
3411                 ASSERT(setsize != 0);
3412                 error = as_iset_default_lpsize(as, setaddr, setsize,
3413                     rflags, rtype);
3414         }

3416         if (error == IE_RETRY) {
3417                 goto again;
3418         } else if (error == IE_NOMEM) {
3419                 error = EAGAIN;
3420         } else if (error == ENOTSUP) {
3421                 error = EINVAL;
3422         } else if (error == EAGAIN) {
3423                 mutex_enter(&as->a_contents);
3424                 if (!AS_ISNOUNMAPWAIT(as)) {
3425                         if (AS_ISUNMAPWAIT(as) == 0) {
3426                                 cv_broadcast(&as->a_cv);
3427                         }
3428                         AS_SETUNMAPWAIT(as);
3429                         AS_LOCK_EXIT(as);
3429                         AS_LOCK_EXIT(as, &as->a_lock);
3430                         while (AS_ISUNMAPWAIT(as)) {
3431                                 cv_wait(&as->a_cv, &as->a_contents);
3432                         }
3433                         mutex_exit(&as->a_contents);
3434                         AS_LOCK_ENTER(as, RW_WRITER);
3434                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3435                 } else {
3436                         /*
3437                          * We may have raced with
3438                          * segvn_reclaim()/segspt_reclaim(). In this case
3439                          * clean nounmapwait flag and retry since softlockcnt
3440                          * in this segment may be already 0.  We don't drop as
3441                          * writer lock so our number of retries without
3442                          * sleeping should be very small. See segvn_reclaim()
3443                          * for more comments.
3444                          */
3445                         AS_CLRNOUNMAPWAIT(as);
3446                         mutex_exit(&as->a_contents);
3447                 }
3448                 goto again;
3449         }

3451         as_setwatch(as);
3452         AS_LOCK_EXIT(as);
3452         AS_LOCK_EXIT(as, &as->a_lock);
3453         return (error);
3454 }

3456 /*
```

```
3457  * Setup all of the uninitialized watched pages that we can.
3458  */
3459 void
3460 as_setwatch(struct as *as)
3461 {
3462         struct watched_page *pwp;
3463         struct seg *seg;
3464         caddr_t vaddr;
3465         uint_t prot;
3466         int  err, retrycnt;

3468         if (avl_numnodes(&as->a_wpage) == 0)
3469                 return;

3471         ASSERT(AS_WRITE_HELD(as));
3471         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3473         for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3474             pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3475                 retrycnt = 0;
3476         retry:
3477                 vaddr = pwp->wp_vaddr;
3478                 if (pwp->wp_oprot != 0 ||       /* already set up */
3479                     (seg = as_segat(as, vaddr)) == NULL ||
3480                     SEGOP_GETPROT(seg, vaddr, 0, &prot) != 0)
3481                         continue;

3483                 pwp->wp_oprot = prot;
3484                 if (pwp->wp_read)
3485                         prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3486                 if (pwp->wp_write)
3487                         prot &= ~PROT_WRITE;
3488                 if (pwp->wp_exec)
3489                         prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3490                 if (!(pwp->wp_flags & WP_NOWATCH) && prot != pwp->wp_oprot) {
3491                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3492                         if (err == IE_RETRY) {
3493                                 pwp->wp_oprot = 0;
3494                                 ASSERT(retrycnt == 0);
3495                                 retrycnt++;
3496                                 goto retry;
3497                         }
3498                 }
3499                 pwp->wp_prot = prot;
3500         }
3501 }

3503 /*
3504  * Clear all of the watched pages in the address space.
3505  */
3506 void
3507 as_clearwatch(struct as *as)
3508 {
3509         struct watched_page *pwp;
3510         struct seg *seg;
3511         caddr_t vaddr;
3512         uint_t prot;
3513         int err, retrycnt;

3515         if (avl_numnodes(&as->a_wpage) == 0)
3516                 return;

3518         ASSERT(AS_WRITE_HELD(as));
3518         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3520         for (pwp = avl_first(&as->a_wpage); pwp != NULL;
```



```
3521             pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3522                 retrycnt = 0;
3523         retry:
3524                 vaddr = pwp->wp_vaddr;
3525                 if (pwp->wp_oprot == 0 ||       /* not set up */
3526                     (seg = as_segat(as, vaddr)) == NULL)
3527                         continue;

3529                 if ((prot = pwp->wp_oprot) != pwp->wp_prot) {
3530                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3531                         if (err == IE_RETRY) {
3532                                 ASSERT(retrycnt == 0);
3533                                 retrycnt++;
3534                                 goto retry;
3535                         }
3536                 }
3537                 pwp->wp_oprot = 0;
3538                 pwp->wp_prot = 0;
3539         }
3540 }

3542 /*
3543  * Force a new setup for all the watched pages in the range.
3544  */
3545 static void
3546 as_setwatchprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
3547 {
3548         struct watched_page *pwp;
3549         struct watched_page tpw;
3550         caddr_t eaddr = addr + size;
3551         caddr_t vaddr;
3552         struct seg *seg;
3553         int err, retrycnt;
3554         uint_t  wprot;
3555         avl_index_t where;

3557         if (avl_numnodes(&as->a_wpage) == 0)
3558                 return;

3560         ASSERT(AS_WRITE_HELD(as));
3560         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3562         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3563         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3564                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3566         while (pwp != NULL && pwp->wp_vaddr < eaddr) {
3567                 retrycnt = 0;
3568                 vaddr = pwp->wp_vaddr;

3570                 wprot = prot;
3571                 if (pwp->wp_read)
3572                         wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3573                 if (pwp->wp_write)
3574                         wprot &= ~PROT_WRITE;
3575                 if (pwp->wp_exec)
3576                         wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3577                 if (!(pwp->wp_flags & WP_NOWATCH) && wprot != pwp->wp_oprot) {
3578                 retry:
3579                         seg = as_segat(as, vaddr);
3580                         if (seg == NULL) {
3581                                 panic("as_setwatchprot: no seg");
3582                                 /*NOTREACHED*/
3583                         }
3584                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, wprot);
3585                         if (err == IE_RETRY) {
```

```
3586                                ASSERT(retrycnt == 0);
3587                                retrycnt++;
3588                                goto retry;
3589                        }
3590                }
3591                pwp->wp_oprot = prot;
3592                pwp->wp_prot = wprot;

3594                pwp = AVL_NEXT(&as->a_wpage, pwp);
3595        }
3596 }

3598 /*
3599  * Clear all of the watched pages in the range.
3600  */
3601 static void
3602 as_clearwatchprot(struct as *as, caddr_t addr, size_t size)
3603 {
3604         caddr_t eaddr = addr + size;
3605         struct watched_page *pwp;
3606         struct watched_page tpw;
3607         uint_t prot;
3608         struct seg *seg;
3609         int err, retrycnt;
3610         avl_index_t where;

3612         if (avl_numnodes(&as->a_wpage) == 0)
3613                 return;

3615         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3616         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3617                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3619         ASSERT(AS_WRITE_HELD(as));
3619         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3621         while (pwp != NULL && pwp->wp_vaddr < eaddr) {

3623                 if ((prot = pwp->wp_oprot) != 0) {
3624                         retrycnt = 0;

3626                         if (prot != pwp->wp_prot) {
3627                         retry:
3628                                 seg = as_segat(as, pwp->wp_vaddr);
3629                                 if (seg == NULL)
3630                                         continue;
3631                                 err = SEGOP_SETPROT(seg, pwp->wp_vaddr,
3632                                     PAGESIZE, prot);
3633                                 if (err == IE_RETRY) {
3634                                         ASSERT(retrycnt == 0);
3635                                         retrycnt++;
3636                                         goto retry;

3638                                 }
3639                         }
3640                         pwp->wp_oprot = 0;
3641                         pwp->wp_prot = 0;
3642                 }

3644                 pwp = AVL_NEXT(&as->a_wpage, pwp);
3645         }
3646 }
```

_____unchanged_portion_omitted_

```
3665 /*
3666  * return memory object ID
```



```
3667  */
3668 int
3669 as_getmemid(struct as *as, caddr_t addr, memid_t *memidp)
3670 {
3671         struct seg      *seg;
3672         int             sts;

3674         AS_LOCK_ENTER(as, RW_READER);
3674         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3675         seg = as_segat(as, addr);
3676         if (seg == NULL) {
3677                 AS_LOCK_EXIT(as);
3677                 AS_LOCK_EXIT(as, &as->a_lock);
3678                 return (EFAULT);
3679         }
3680         /*
3681          * catch old drivers which may not support getmemid
3682          */
3683         if (seg->s_ops->getmemid == NULL) {
3684                 AS_LOCK_EXIT(as);
3684                 AS_LOCK_EXIT(as, &as->a_lock);
3685                 return (ENODEV);
3686         }

3688         sts = SEGOP_GETMEMID(seg, addr, memidp);

3690         AS_LOCK_EXIT(as);
3690         AS_LOCK_EXIT(as, &as->a_lock);
3691         return (sts);
3692 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  187216 Wed Nov 25 13:59:41 2015
new/usr/src/uts/common/vm/vm_page.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

5361 /*
5362  * Mark any existing pages for migration in the given range
5363  */
5364 void
5365 page_mark_migrate(struct seg *seg, caddr_t addr, size_t len,
5366     struct anon_map *amp, ulong_t anon_index, vnode_t *vp,
5367     u_offset_t vnoff, int rflag)
5368 {
5369         struct anon     *ap;
5370         vnode_t         *curvp;
5371         lgrp_t          *from;
5372         pgcnt_t         nlocked;
5373         u_offset_t      off;
5374         pfn_t           pfn;
5375         size_t          pgsz;
5376         size_t          segpgsz;
5377         pgcnt_t         pages;
5378         uint_t          pszc;
5379         page_t          *pp0, *pp;
5380         caddr_t         va;
5381         ulong_t         an_idx;
5382         anon_sync_obj_t cookie;

5384         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
5384         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

5386         /*
5387          * Don't do anything if don't need to do lgroup optimizations
5388          * on this system
5389          */
5390         if (!lgrp_optimizations())
5391                 return;

5393         /*
5394          * Align address and length to (potentially large) page boundary
5395          */
5396         segpgsz = page_get_pagesize(seg->s_szc);
5397         addr = (caddr_t)P2ALIGN((uintptr_t)addr, segpgsz);
5398         if (rflag)
5399                 len = P2ROUNDUP(len, segpgsz);

5401         /*
5402          * Do one (large) page at a time
5403          */
5404         va = addr;
5405         while (va < addr + len) {
5406                 /*
5407                  * Lookup (root) page for vnode and offset corresponding to
5408                  * this virtual address
5409                  * Try anonmap first since there may be copy-on-write
5410                  * pages, but initialize vnode pointer and offset using
5411                  * vnode arguments just in case there isn't an amp.
5412                  */
5413                 curvp = vp;
5414                 off = vnoff + va - seg->s_base;
5415                 if (amp) {
5416                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5417                         an_idx = anon_index + seg_page(seg, va);
5418                         anon_array_enter(amp, an_idx, &cookie);
```



```
5419                         ap = anon_get_ptr(amp->ahp, an_idx);
5420                         if (ap)
5421                                 swap_xlate(ap, &curvp, &off);
5422                         anon_array_exit(&cookie);
5423                         ANON_LOCK_EXIT(&amp->a_rwlock);
5424                 }

5426                 pp = NULL;
5427                 if (curvp)
5428                         pp = page_lookup(curvp, off, SE_SHARED);

5430                 /*
5431                  * If there isn't a page at this virtual address,
5432                  * skip to next page
5433                  */
5434                 if (pp == NULL) {
5435                         va += PAGESIZE;
5436                         continue;
5437                 }

5439                 /*
5440                  * Figure out which lgroup this page is in for kstats
5441                  */
5442                 pfn = page_pptonum(pp);
5443                 from = lgrp_pfn_to_lgrp(pfn);

5445                 /*
5446                  * Get page size, and round up and skip to next page boundary
5447                  * if unaligned address
5448                  */
5449                 pszc = pp->p_szc;
5450                 pgsz = page_get_pagesize(pszc);
5451                 pages = btop(pgsz);
5452                 if (!IS_P2ALIGNED(va, pgsz) ||
5453                     !IS_P2ALIGNED(pfn, pages) ||
5454                     pgsz > segpgsz) {
5455                         pgsz = MIN(pgsz, segpgsz);
5456                         page_unlock(pp);
5457                         pages = btop(P2END((uintptr_t)va, pgsz) -
5458                             (uintptr_t)va);
5459                         va = (caddr_t)P2END((uintptr_t)va, pgsz);
5460                         lgrp_stat_add(from->lgrp_id, LGRP_PMM_FAIL_PGS, pages);
5461                         continue;
5462                 }

5464                 /*
5465                  * Upgrade to exclusive lock on page
5466                  */
5467                 if (!page_tryupgrade(pp)) {
5468                         page_unlock(pp);
5469                         va += pgsz;
5470                         lgrp_stat_add(from->lgrp_id, LGRP_PMM_FAIL_PGS,
5471                             btop(pgsz));
5472                         continue;
5473                 }

5475                 pp0 = pp++;
5476                 nlocked = 1;

5478                 /*
5479                  * Lock constituent pages if this is large page
5480                  */
5481                 if (pages > 1) {
5482                         /*
5483                          * Lock all constituents except root page, since it
5484                          * should be locked already.
```

```
5485                          */
5486                         for (; nlocked < pages; nlocked++) {
5487                                 if (!page_trylock(pp, SE_EXCL)) {
5488                                         break;
5489                                 }
5490                                 if (PP_ISFREE(pp) ||
5491                                     pp->p_szc != pszc) {
5492                                         /*
5493                                          * hat_page_demote() raced in with us.
5494                                          */
5495                                         ASSERT(!IS_SWAPFSVP(curvp));
5496                                         page_unlock(pp);
5497                                         break;
5498                                 }
5499                                 pp++;
5500                         }
5501                 }

5503                 /*
5504                  * If all constituent pages couldn't be locked,
5505                  * unlock pages locked so far and skip to next page.
5506                  */
5507                 if (nlocked < pages) {
5508                         while (pp0 < pp) {
5509                                 page_unlock(pp0++);
5510                         }
5511                         va += pgsz;
5512                         lgrp_stat_add(from->lgrp_id, LGRP_PMM_FAIL_PGS,
5513                             btop(pgsz));
5514                         continue;
5515                 }

5517                 /*
5518                  * hat_page_demote() can no longer happen
5519                  * since last cons page had the right p_szc after
5520                  * all cons pages were locked. all cons pages
5521                  * should now have the same p_szc.
5522                  */

5524                 /*
5525                  * All constituent pages locked successfully, so mark
5526                  * large page for migration and unload the mappings of
5527                  * constituent pages, so a fault will occur on any part of the
5528                  * large page
5529                  */
5530                 PP_SETMIGRATE(pp0);
5531                 while (pp0 < pp) {
5532                         (void) hat_pageunload(pp0, HAT_FORCE_PGUNLOAD);
5533                         ASSERT(hat_page_getshare(pp0) == 0);
5534                         page_unlock(pp0++);
5535                 }
5536                 lgrp_stat_add(from->lgrp_id, LGRP_PMM_PGS, nlocked);

5538                 va += pgsz;
5539         }
5540 }

5542 /*
5543  * Migrate any pages that have been marked for migration in the given range
5544  */
5545 void
5546 page_migrate(
5547         struct seg      *seg,
5548         caddr_t         addr,
5549         page_t          **ppa,
5550         pgcnt_t         npages)
```



```
5551 {
5552         lgrp_t          *from;
5553         lgrp_t          *to;
5554         page_t          *newpp;
5555         page_t          *pp;
5556         pfn_t           pfn;
5557         size_t          pgsz;
5558         spgcnt_t        page_cnt;
5559         spgcnt_t        i;
5560         uint_t          pszc;

5562         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
5562         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

5564         while (npages > 0) {
5565                 pp = *ppa;
5566                 pszc = pp->p_szc;
5567                 pgsz = page_get_pagesize(pszc);
5568                 page_cnt = btop(pgsz);

5570                 /*
5571                  * Check to see whether this page is marked for migration
5572                  *
5573                  * Assume that root page of large page is marked for
5574                  * migration and none of the other constituent pages
5575                  * are marked.  This really simplifies clearing the
5576                  * migrate bit by not having to clear it from each
5577                  * constituent page.
5578                  *
5579                  * note we don't want to relocate an entire large page if
5580                  * someone is only using one subpage.
5581                  */
5582                 if (npages < page_cnt)
5583                         break;

5585                 /*
5586                  * Is it marked for migration?
5587                  */
5588                 if (!PP_ISMIGRATE(pp))
5589                         goto next;

5591                 /*
5592                  * Determine lgroups that page is being migrated between
5593                  */
5594                 pfn = page_pptonum(pp);
5595                 if (!IS_P2ALIGNED(pfn, page_cnt)) {
5596                         break;
5597                 }
5598                 from = lgrp_pfn_to_lgrp(pfn);
5599                 to = lgrp_mem_choose(seg, addr, pgsz);

5601                 /*
5602                  * Need to get exclusive lock's to migrate
5603                  */
5604                 for (i = 0; i < page_cnt; i++) {
5605                         ASSERT(PAGE_LOCKED(ppa[i]));
5606                         if (page_pptonum(ppa[i]) != pfn + i ||
5607                             ppa[i]->p_szc != pszc) {
5608                                 break;
5609                         }
5610                         if (!page_tryupgrade(ppa[i])) {
5611                                 lgrp_stat_add(from->lgrp_id,
5612                                     LGRP_PM_FAIL_LOCK_PGS,
5613                                     page_cnt);
5614                                 break;
5615                         }
```

```
5617                         /*
5618                          * Check to see whether we are trying to migrate
5619                          * page to lgroup where it is allocated already.
5620                          * If so, clear the migrate bit and skip to next
5621                          * page.
5622                          */
5623                         if (i == 0 && to == from) {
5624                                 PP_CLRMIGRATE(ppa[0]);
5625                                 page_downgrade(ppa[0]);
5626                                 goto next;
5627                         }
5628                 }

5630                 /*
5631                  * If all constituent pages couldn't be locked,
5632                  * unlock pages locked so far and skip to next page.
5633                  */
5634                 if (i != page_cnt) {
5635                         while (--i != -1) {
5636                                 page_downgrade(ppa[i]);
5637                         }
5638                         goto next;
5639                 }

5641                 (void) page_create_wait(page_cnt, PG_WAIT);
5642                 newpp = page_get_replacement_page(pp, to, PGR_SAMESZC);
5643                 if (newpp == NULL) {
5644                         page_create_putback(page_cnt);
5645                         for (i = 0; i < page_cnt; i++) {
5646                                 page_downgrade(ppa[i]);
5647                         }
5648                         lgrp_stat_add(to->lgrp_id, LGRP_PM_FAIL_ALLOC_PGS,
5649                             page_cnt);
5650                         goto next;
5651                 }
5652                 ASSERT(newpp->p_szc == pszc);
5653                 /*
5654                  * Clear migrate bit and relocate page
5655                  */
5656                 PP_CLRMIGRATE(pp);
5657                 if (page_relocate(&pp, &newpp, 0, 1, &page_cnt, to)) {
5658                         panic("page_migrate: page_relocate failed");
5659                 }
5660                 ASSERT(page_cnt * PAGESIZE == pgsz);

5662                 /*
5663                  * Keep stats for number of pages migrated from and to
5664                  * each lgroup
5665                  */
5666                 lgrp_stat_add(from->lgrp_id, LGRP_PM_SRC_PGS, page_cnt);
5667                 lgrp_stat_add(to->lgrp_id, LGRP_PM_DEST_PGS, page_cnt);
5668                 /*
5669                  * update the page_t array we were passed in and
5670                  * unlink constituent pages of a large page.
5671                  */
5672                 for (i = 0; i < page_cnt; ++i, ++pp) {
5673                         ASSERT(PAGE_EXCL(newpp));
5674                         ASSERT(newpp->p_szc == pszc);
5675                         ppa[i] = newpp;
5676                         pp = newpp;
5677                         page_sub(&newpp, pp);
5678                         page_downgrade(pp);
5679                 }
5680                 ASSERT(newpp == NULL);
5681 next:
```



```
5682                 addr += pgsz;
5683                 ppa += page_cnt;
5684                 npages -= page_cnt;
5685         }
5686 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   51450 Wed Nov 25 13:59:41 2015
new/usr/src/uts/common/vm/vm_seg.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1636 /*
1637  * Unmap a segment and free it from its associated address space.
1638  * This should be called by anybody who's finished with a whole segment's
1639  * mapping.  Just calls SEGOP_UNMAP() on the whole mapping .  It is the
1640  * responsibility of the segment driver to unlink the the segment
1641  * from the address space, and to free public and private data structures
1642  * associated with the segment.  (This is typically done by a call to
1643  * seg_free()).
1644  */
1645 void
1646 seg_unmap(struct seg *seg)
1647 {
1648 #ifdef DEBUG
1649         int ret;
1650 #endif /* DEBUG */

1652         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
1652         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1654         /* Shouldn't have called seg_unmap if mapping isn't yet established */
1655         ASSERT(seg->s_data != NULL);

1657         /* Unmap the whole mapping */
1658 #ifdef DEBUG
1659         ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1660         ASSERT(ret == 0);
1661 #else
1662         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1663 #endif /* DEBUG */
1664 }
______unchanged_portion_omitted_
```

```
**********************************************************
   57898 Wed Nov 25 13:59:41 2015
new/usr/src/uts/common/vm/vm_usage.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1442 /*
1443  * Based on the current calculation flags, find the relevant entities
1444  * which are relative to the process.  Then calculate each segment
1445  * in the process'es address space for each relevant entity.
1446  */
1447 static void
1448 vmu_calculate_proc(proc_t *p)
1449 {
1450         vmu_entity_t *entities = NULL;
1451         vmu_zone_t *zone;
1452         vmu_entity_t *tmp;
1453         struct as *as;
1454         struct seg *seg;
1455         int ret;

1457         /* Figure out which entities are being computed */
1458         if ((vmu_data.vmu_system) != NULL) {
1459                 tmp = vmu_data.vmu_system;
1460                 tmp->vme_next_calc = entities;
1461                 entities = tmp;
1462         }
1463         if (vmu_data.vmu_calc_flags &
1464             (VMUSAGE_ZONE | VMUSAGE_ALL_ZONES | VMUSAGE_PROJECTS |
1465             VMUSAGE_ALL_PROJECTS | VMUSAGE_TASKS | VMUSAGE_ALL_TASKS |
1466             VMUSAGE_RUSERS | VMUSAGE_ALL_RUSERS | VMUSAGE_EUSERS |
1467             VMUSAGE_ALL_EUSERS)) {
1468                 ret = i_mod_hash_find_nosync(vmu_data.vmu_zones_hash,
1469                     (mod_hash_key_t)(uintptr_t)p->p_zone->zone_id,
1470                     (mod_hash_val_t *)&zone);
1471                 if (ret != 0) {
1472                         zone = vmu_alloc_zone(p->p_zone->zone_id);
1473                         ret = i_mod_hash_insert_nosync(vmu_data.vmu_zones_hash,
1474                             (mod_hash_key_t)(uintptr_t)p->p_zone->zone_id,
1475                             (mod_hash_val_t)zone, (mod_hash_hndl_t)0);
1476                         ASSERT(ret == 0);
1477                 }
1478                 if (zone->vmz_zone != NULL) {
1479                         tmp = zone->vmz_zone;
1480                         tmp->vme_next_calc = entities;
1481                         entities = tmp;
1482                 }
1483                 if (vmu_data.vmu_calc_flags &
1484                     (VMUSAGE_PROJECTS | VMUSAGE_ALL_PROJECTS)) {
1485                         tmp = vmu_find_insert_entity(zone->vmz_projects_hash,
1486                             p->p_task->tk_proj->kpj_id, VMUSAGE_PROJECTS,
1487                             zone->vmz_id);
1488                         tmp->vme_next_calc = entities;
1489                         entities = tmp;
1490                 }
1491                 if (vmu_data.vmu_calc_flags &
1492                     (VMUSAGE_TASKS | VMUSAGE_ALL_TASKS)) {
1493                         tmp = vmu_find_insert_entity(zone->vmz_tasks_hash,
1494                             p->p_task->tk_tkid, VMUSAGE_TASKS, zone->vmz_id);
1495                         tmp->vme_next_calc = entities;
1496                         entities = tmp;
1497                 }
1498                 if (vmu_data.vmu_calc_flags &
1499                     (VMUSAGE_RUSERS | VMUSAGE_ALL_RUSERS)) {
1500                         tmp = vmu_find_insert_entity(zone->vmz_rusers_hash,
```



```
1501                             crgetruid(p->p_cred), VMUSAGE_RUSERS, zone->vmz_id);
1502                         tmp->vme_next_calc = entities;
1503                         entities = tmp;
1504                 }
1505                 if (vmu_data.vmu_calc_flags &
1506                     (VMUSAGE_EUSERS | VMUSAGE_ALL_EUSERS)) {
1507                         tmp = vmu_find_insert_entity(zone->vmz_eusers_hash,
1508                             crgetuid(p->p_cred), VMUSAGE_EUSERS, zone->vmz_id);
1509                         tmp->vme_next_calc = entities;
1510                         entities = tmp;
1511                 }
1512         }
1513         /* Entities which collapse projects and users for all zones */
1514         if (vmu_data.vmu_calc_flags & VMUSAGE_COL_PROJECTS) {
1515                 tmp = vmu_find_insert_entity(vmu_data.vmu_projects_col_hash,
1516                     p->p_task->tk_proj->kpj_id, VMUSAGE_PROJECTS, ALL_ZONES);
1517                 tmp->vme_next_calc = entities;
1518                 entities = tmp;
1519         }
1520         if (vmu_data.vmu_calc_flags & VMUSAGE_COL_RUSERS) {
1521                 tmp = vmu_find_insert_entity(vmu_data.vmu_rusers_col_hash,
1522                     crgetruid(p->p_cred), VMUSAGE_RUSERS, ALL_ZONES);
1523                 tmp->vme_next_calc = entities;
1524                 entities = tmp;
1525         }
1526         if (vmu_data.vmu_calc_flags & VMUSAGE_COL_EUSERS) {
1527                 tmp = vmu_find_insert_entity(vmu_data.vmu_eusers_col_hash,
1528                     crgetuid(p->p_cred), VMUSAGE_EUSERS, ALL_ZONES);
1529                 tmp->vme_next_calc = entities;
1530                 entities = tmp;
1531         }

1533         ASSERT(entities != NULL);
1534         /* process all segs in process's address space */
1535         as = p->p_as;
1536         AS_LOCK_ENTER(as, RW_READER);
1536         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1537         for (seg = AS_SEGFIRST(as); seg != NULL;
1538             seg = AS_SEGNEXT(as, seg)) {
1539                 vmu_calculate_seg(entities, seg);
1540         }
1541         AS_LOCK_EXIT(as);
1541         AS_LOCK_EXIT(as, &as->a_lock);
1542 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36908 Wed Nov 25 13:59:41 2015
new/usr/src/uts/common/xen/io/xpvtap.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_


 770 /*
 771  * xpvtap_segmf_register()
 772  */
 773 static int
 774 xpvtap_segmf_register(xpvtap_state_t *state)
 775 {
 776         struct seg *seg;
 777         uint64_t pte_ma;
 778         struct as *as;
 779         caddr_t uaddr;
 780         uint_t pgcnt;
 781         int i;


 784         as = state->bt_map.um_as;
 785         pgcnt = btopr(state->bt_map.um_guest_size);
 786         uaddr = state->bt_map.um_guest_pages;

 788         if (pgcnt == 0) {
 789                 return (DDI_FAILURE);
 790         }

 792         AS_LOCK_ENTER(as, RW_READER);
 792         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

 794         seg = as_findseg(as, state->bt_map.um_guest_pages, 0);
 795         if ((seg == NULL) || ((uaddr + state->bt_map.um_guest_size) >
 796             (seg->s_base + seg->s_size))) {
 797                 AS_LOCK_EXIT(as);
 797                 AS_LOCK_EXIT(as, &as->a_lock);
 798                 return (DDI_FAILURE);
 799         }

 801         /*
 802          * lock down the htables so the HAT can't steal them. Register the
 803          * PTE MA's for each gref page with seg_mf so we can do user space
 804          * gref mappings.
 805          */
 806         for (i = 0; i < pgcnt; i++) {
 807                 hat_prepare_mapping(as->a_hat, uaddr, &pte_ma);
 808                 hat_devload(as->a_hat, uaddr, PAGESIZE, (pfn_t)0,
 809                     PROT_READ | PROT_WRITE | PROT_USER | HAT_UNORDERED_OK,
 810                     HAT_LOAD_NOCONSIST | HAT_LOAD_LOCK);
 811                 hat_release_mapping(as->a_hat, uaddr);
 812                 segmf_add_gref_pte(seg, uaddr, pte_ma);
 813                 uaddr += PAGESIZE;
 814         }

 816         state->bt_map.um_registered = B_TRUE;

 818         AS_LOCK_EXIT(as);
 818         AS_LOCK_EXIT(as, &as->a_lock);

 820         return (DDI_SUCCESS);
 821 }


 824 /*
```



```
 825  * xpvtap_segmf_unregister()
 826  *    as_callback routine
 827  */
 828 /*ARGSUSED*/
 829 static void
 830 xpvtap_segmf_unregister(struct as *as, void *arg, uint_t event)
 831 {
 832         xpvtap_state_t *state;
 833         caddr_t uaddr;
 834         uint_t pgcnt;
 835         int i;


 838         state = (xpvtap_state_t *)arg;
 839         if (!state->bt_map.um_registered) {
 840                 /* remove the callback (which is this routine) */
 841                 (void) as_delete_callback(as, arg);
 842                 return;
 843         }

 845         pgcnt = btopr(state->bt_map.um_guest_size);
 846         uaddr = state->bt_map.um_guest_pages;

 848         /* unmap any outstanding req's grefs */
 849         xpvtap_rs_flush(state->bt_map.um_rs, xpvtap_user_request_unmap, state);

 851         /* Unlock the gref pages */
 852         for (i = 0; i < pgcnt; i++) {
 853                 AS_LOCK_ENTER(as, RW_WRITER);
 853                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 854                 hat_prepare_mapping(as->a_hat, uaddr, NULL);
 855                 hat_unload(as->a_hat, uaddr, PAGESIZE, HAT_UNLOAD_UNLOCK);
 856                 hat_release_mapping(as->a_hat, uaddr);
 857                 AS_LOCK_EXIT(as);
 857                 AS_LOCK_EXIT(as, &as->a_lock);
 858                 uaddr += PAGESIZE;
 859         }

 861         /* remove the callback (which is this routine) */
 862         (void) as_delete_callback(as, arg);

 864         state->bt_map.um_registered = B_FALSE;
 865 }
______unchanged_portion_omitted_


1182 /*
1183  * xpvtap_user_request_map()
1184  */
1185 static int
1186 xpvtap_user_request_map(xpvtap_state_t *state, blkif_request_t *req,
1187     uint_t *uid)
1188 {
1189         grant_ref_t gref[BLKIF_MAX_SEGMENTS_PER_REQUEST];
1190         struct seg *seg;
1191         struct as *as;
1192         domid_t domid;
1193         caddr_t uaddr;
1194         uint_t flags;
1195         int i;
1196         int e;


1199         domid = xvdi_get_oeid(state->bt_dip);

1201         as = state->bt_map.um_as;
```

```
1202         if ((as == NULL) || (state->bt_map.um_guest_pages == NULL)) {
1203                 return (DDI_FAILURE);
1204         }

1206         /* has to happen after segmap returns */
1207         if (!state->bt_map.um_registered) {
1208                 /* register the pte's with segmf */
1209                 e = xpvtap_segmf_register(state);
1210                 if (e != DDI_SUCCESS) {
1211                         return (DDI_FAILURE);
1212                 }
1213         }

1215         /* alloc an ID for the user ring */
1216         e = xpvtap_rs_alloc(state->bt_map.um_rs, uid);
1217         if (e != DDI_SUCCESS) {
1218                 return (DDI_FAILURE);
1219         }

1221         /* if we don't have any segments to map, we're done */
1222         if ((req->operation == BLKIF_OP_WRITE_BARRIER) ||
1223             (req->operation == BLKIF_OP_FLUSH_DISKCACHE) ||
1224             (req->nr_segments == 0)) {
1225                 return (DDI_SUCCESS);
1226         }

1228         /* get the apps gref address */
1229         uaddr = XPVTAP_GREF_REQADDR(state->bt_map.um_guest_pages, *uid);

1231         AS_LOCK_ENTER(as, RW_READER);
1231         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1232         seg = as_findseg(as, state->bt_map.um_guest_pages, 0);
1233         if ((seg == NULL) || ((uaddr + mmu_ptob(req->nr_segments)) >
1234             (seg->s_base + seg->s_size))) {
1235                 AS_LOCK_EXIT(as);
1235                 AS_LOCK_EXIT(as, &as->a_lock);
1236                 return (DDI_FAILURE);
1237         }

1239         /* if we are reading from disk, we are writing into memory */
1240         flags = 0;
1241         if (req->operation == BLKIF_OP_READ) {
1242                 flags |= SEGMF_GREF_WR;
1243         }

1245         /* Load the grefs into seg_mf */
1246         for (i = 0; i < req->nr_segments; i++) {
1247                 gref[i] = req->seg[i].gref;
1248         }
1249         (void) segmf_add_grefs(seg, uaddr, flags, gref, req->nr_segments,
1250             domid);

1252         AS_LOCK_EXIT(as);
1252         AS_LOCK_EXIT(as, &as->a_lock);

1254         return (DDI_SUCCESS);
1255 }
```

_____***unchanged_portion_omitted_***

```
1294 static void
1295 xpvtap_user_request_unmap(xpvtap_state_t *state, uint_t uid)
1296 {
1297         blkif_request_t *req;
1298         struct seg *seg;
1299         struct as *as;
```



```
1300         caddr_t uaddr;
1301         int e;


1304         as = state->bt_map.um_as;
1305         if (as == NULL) {
1306                 return;
1307         }

1309         /* get a copy of the original request */
1310         req = &state->bt_map.um_outstanding_reqs[uid];

1312         /* unmap the grefs for this request */
1313         if ((req->operation != BLKIF_OP_WRITE_BARRIER) &&
1314             (req->operation != BLKIF_OP_FLUSH_DISKCACHE) &&
1315             (req->nr_segments != 0)) {
1316                 uaddr = XPVTAP_GREF_REQADDR(state->bt_map.um_guest_pages, uid);
1317                 AS_LOCK_ENTER(as, RW_READER);
1317                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1318                 seg = as_findseg(as, state->bt_map.um_guest_pages, 0);
1319                 if ((seg == NULL) || ((uaddr + mmu_ptob(req->nr_segments)) >
1320                     (seg->s_base + seg->s_size))) {
1321                         AS_LOCK_EXIT(as);
1321                         AS_LOCK_EXIT(as, &as->a_lock);
1322                         xpvtap_rs_free(state->bt_map.um_rs, uid);
1323                         return;
1324                 }

1326                 e = segmf_release_grefs(seg, uaddr, req->nr_segments);
1327                 if (e != 0) {
1328                         cmn_err(CE_WARN, "unable to release grefs");
1329                 }

1331                 AS_LOCK_EXIT(as);
1331                 AS_LOCK_EXIT(as, &as->a_lock);
1332         }

1334         /* free up the user ring id */
1335         xpvtap_rs_free(state->bt_map.um_rs, uid);
1336 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142099 Wed Nov 25 13:59:41 2015
new/usr/src/uts/i86pc/io/rootnex.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_


 973 /*
 974  * rootnex_map_fault()
 975  *
 976  *      fault in mappings for requestors
 977  */
 978 /*ARGSUSED*/
 979 static int
 980 rootnex_map_fault(dev_info_t *dip, dev_info_t *rdip, struct hat *hat,
 981     struct seg *seg, caddr_t addr, struct devpage *dp, pfn_t pfn, uint_t prot,
 982     uint_t lock)
 983 {

 985 #ifdef  DDI_MAP_DEBUG
 986         ddi_map_debug("rootnex_map_fault: address <%x> pfn <%x>", addr, pfn);
 987         ddi_map_debug(" Seg <%s>\n",
 988             seg->s_ops == &segdev_ops ? "segdev" :
 989             seg == &kvseg ? "segkmem" : "NONE!");
 990 #endif  /* DDI_MAP_DEBUG */

 992         /*
 993          * This is all terribly broken, but it is a start
 994          *
 995          * XXX  Note that this test means that segdev_ops
 996          *      must be exported from seg_dev.c.
 997          * XXX  What about devices with their own segment drivers?
 998          */
 999         if (seg->s_ops == &segdev_ops) {
1000                 struct segdev_data *sdp = (struct segdev_data *)seg->s_data;

1002                 if (hat == NULL) {
1003                         /*
1004                          * This is one plausible interpretation of
1005                          * a null hat i.e. use the first hat on the
1006                          * address space hat list which by convention is
1007                          * the hat of the system MMU.  At alternative
1008                          * would be to panic .. this might well be better ..
1009                          */
1010                         ASSERT(AS_READ_HELD(seg->s_as));
1010                         ASSERT(AS_READ_HELD(seg->s_as, &seg->s_as->a_lock));
1011                         hat = seg->s_as->a_hat;
1012                         cmn_err(CE_NOTE, "rootnex_map_fault: nil hat");
1013                 }
1014                 hat_devload(hat, addr, MMU_PAGESIZE, pfn, prot | sdp->hat_attr,
1015                     (lock ? HAT_LOAD_LOCK : HAT_LOAD));
1016         } else if (seg == &kvseg && dp == NULL) {
1017                 hat_devload(kas.a_hat, addr, MMU_PAGESIZE, pfn, prot,
1018                     HAT_LOAD_LOCK);
1019         } else
1020                 return (DDI_FAILURE);
1021         return (DDI_SUCCESS);
1022 }
______unchanged_portion_omitted_
```

```
**********************************************************
  107401 Wed Nov 25 13:59:42 2015
new/usr/src/uts/i86pc/vm/hat_i86.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 238 /*
 239  * Allocate a hat structure for as. We also create the top level
 240  * htable and initialize it to contain the kernel hat entries.
 241  */
 242 hat_t *
 243 hat_alloc(struct as *as)
 244 {
 245         hat_t                   *hat;
 246         htable_t                *ht;    /* top level htable */
 247         uint_t                  use_vlp;
 248         uint_t                  r;
 249         hat_kernel_range_t      *rp;
 250         uintptr_t               va;
 251         uintptr_t               eva;
 252         uint_t                  start;
 253         uint_t                  cnt;
 254         htable_t                *src;

 256         /*
 257          * Once we start creating user process HATs we can enable
 258          * the htable_steal() code.
 259          */
 260         if (can_steal_post_boot == 0)
 261                 can_steal_post_boot = 1;

 263         ASSERT(AS_WRITE_HELD(as));
 263         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
 264         hat = kmem_cache_alloc(hat_cache, KM_SLEEP);
 265         hat->hat_as = as;
 266         mutex_init(&hat->hat_mutex, NULL, MUTEX_DEFAULT, NULL);
 267         ASSERT(hat->hat_flags == 0);

 269 #if defined(__xpv)
 270         /*
 271          * No VLP stuff on the hypervisor due to the 64-bit split top level
 272          * page tables.  On 32-bit it's not needed as the hypervisor takes
 273          * care of copying the top level PTEs to a below 4Gig page.
 274          */
 275         use_vlp = 0;
 276 #else   /* __xpv */
 277         /* 32 bit processes uses a VLP style hat when running with PAE */
 278 #if defined(__amd64)
 279         use_vlp = (ttoproc(curthread)->p_model == DATAMODEL_ILP32);
 280 #elif defined(__i386)
 281         use_vlp = mmu.pae_hat;
 282 #endif
 283 #endif  /* __xpv */
 284         if (use_vlp) {
 285                 hat->hat_flags = HAT_VLP;
 286                 bzero(hat->hat_vlp_ptes, VLP_SIZE);
 287         }

 289         /*
 290          * Allocate the htable hash
 291          */
 292         if ((hat->hat_flags & HAT_VLP)) {
 293                 hat->hat_num_hash = mmu.vlp_hash_cnt;
 294                 hat->hat_ht_hash = kmem_cache_alloc(vlp_hash_cache, KM_SLEEP);
 295         } else {
```



```
 296                 hat->hat_num_hash = mmu.hash_cnt;
 297                 hat->hat_ht_hash = kmem_cache_alloc(hat_hash_cache, KM_SLEEP);
 298         }
 299         bzero(hat->hat_ht_hash, hat->hat_num_hash * sizeof (htable_t *));

 301         /*
 302          * Initialize Kernel HAT entries at the top of the top level page
 303          * tables for the new hat.
 304          */
 305         hat->hat_htable = NULL;
 306         hat->hat_ht_cached = NULL;
 307         XPV_DISALLOW_MIGRATE();
 308         ht = htable_create(hat, (uintptr_t)0, TOP_LEVEL(hat), NULL);
 309         hat->hat_htable = ht;

 311 #if defined(__amd64)
 312         if (hat->hat_flags & HAT_VLP)
 313                 goto init_done;
 314 #endif

 316         for (r = 0; r < num_kernel_ranges; ++r) {
 317                 rp = &kernel_ranges[r];
 318                 for (va = rp->hkr_start_va; va != rp->hkr_end_va;
 319                     va += cnt * LEVEL_SIZE(rp->hkr_level)) {

 321                         if (rp->hkr_level == TOP_LEVEL(hat))
 322                                 ht = hat->hat_htable;
 323                         else
 324                                 ht = htable_create(hat, va, rp->hkr_level,
 325                                     NULL);

 327                         start = htable_va2entry(va, ht);
 328                         cnt = HTABLE_NUM_PTES(ht) - start;
 329                         eva = va +
 330                             ((uintptr_t)cnt << LEVEL_SHIFT(rp->hkr_level));
 331                         if (rp->hkr_end_va != 0 &&
 332                             (eva > rp->hkr_end_va || eva == 0))
 333                                 cnt = htable_va2entry(rp->hkr_end_va, ht) -
 334                                     start;

 336 #if defined(__i386) && !defined(__xpv)
 337                         if (ht->ht_flags & HTABLE_VLP) {
 338                                 bcopy(&vlp_page[start],
 339                                     &hat->hat_vlp_ptes[start],
 340                                     cnt * sizeof (x86pte_t));
 341                                 continue;
 342                         }
 343 #endif
 344                         src = htable_lookup(kas.a_hat, va, rp->hkr_level);
 345                         ASSERT(src != NULL);
 346                         x86pte_copy(src, ht, start, cnt);
 347                         htable_release(src);
 348                 }
 349         }

 351 init_done:

 353 #if defined(__xpv)
 354         /*
 355          * Pin top level page tables after initializing them
 356          */
 357         xen_pin(hat->hat_htable->ht_pfn, mmu.max_level);
 358 #if defined(__amd64)
 359         xen_pin(hat->hat_user_ptable, mmu.max_level);
 360 #endif
 361 #endif
```

```
362         XPV_ALLOW_MIGRATE();

364         /*
365          * Put it at the start of the global list of all hats (used by stealing)
366          *
367          * kas.a_hat is not in the list but is instead used to find the
368          * first and last items in the list.
369          *
370          * - kas.a_hat->hat_next points to the start of the user hats.
371          *   The list ends where hat->hat_next == NULL
372          *
373          * - kas.a_hat->hat_prev points to the last of the user hats.
374          *   The list begins where hat->hat_prev == NULL
375          */
376         mutex_enter(&hat_list_lock);
377         hat->hat_prev = NULL;
378         hat->hat_next = kas.a_hat->hat_next;
379         if (hat->hat_next)
380                 hat->hat_next->hat_prev = hat;
381         else
382                 kas.a_hat->hat_prev = hat;
383         kas.a_hat->hat_next = hat;
384         mutex_exit(&hat_list_lock);

386         return (hat);
387 }

389 /*
390  * process has finished executing but as has not been cleaned up yet.
391  */
392 /*ARGSUSED*/
393 void
394 hat_free_start(hat_t *hat)
395 {
396         ASSERT(AS_WRITE_HELD(hat->hat_as));
396         ASSERT(AS_WRITE_HELD(hat->hat_as, &hat->hat_as->a_lock));

398         /*
399          * If the hat is currently a stealing victim, wait for the stealing
400          * to finish.  Once we mark it as HAT_FREEING, htable_steal()
401          * won't look at its pagetables anymore.
402          */
403         mutex_enter(&hat_list_lock);
404         while (hat->hat_flags & HAT_VICTIM)
405                 cv_wait(&hat_list_cv, &hat_list_lock);
406         hat->hat_flags |= HAT_FREEING;
407         mutex_exit(&hat_list_lock);
408 }
_____unchanged_portion_omitted_


682 /*
683  * initialize hat data structures
684  */
685 void
686 hat_init()
687 {
688 #if defined(__i386)
689         /*
690          * _userlimit must be aligned correctly
691          */
692         if ((_userlimit & LEVEL_MASK(1)) != _userlimit) {
693                 prom_printf("hat_init(): _userlimit=%p, not aligned at %p\n",
694                     (void *)_userlimit, (void *)LEVEL_SIZE(1));
695                 halt("hat_init(): Unable to continue");
696         }
```



```
697 #endif

699         cv_init(&hat_list_cv, NULL, CV_DEFAULT, NULL);

701         /*
702          * initialize kmem caches
703          */
704         htable_init();
705         hment_init();

707         hat_cache = kmem_cache_create("hat_t",
708             sizeof (hat_t), 0, hati_constructor, NULL, NULL,
709             NULL, 0, 0);

711         hat_hash_cache = kmem_cache_create("HatHash",
712             mmu.hash_cnt * sizeof (htable_t *), 0, NULL, NULL, NULL,
713             NULL, 0, 0);

715         /*
716          * VLP hats can use a smaller hash table size on large memroy machines
717          */
718         if (mmu.hash_cnt == mmu.vlp_hash_cnt) {
719                 vlp_hash_cache = hat_hash_cache;
720         } else {
721                 vlp_hash_cache = kmem_cache_create("HatVlpHash",
722                     mmu.vlp_hash_cnt * sizeof (htable_t *), 0, NULL, NULL, NULL,
723                     NULL, 0, 0);
724         }

726         /*
727          * Set up the kernel's hat
728          */
729         AS_LOCK_ENTER(&kas, RW_WRITER);
729         AS_LOCK_ENTER(&kas, &kas.a_lock, RW_WRITER);
730         kas.a_hat = kmem_cache_alloc(hat_cache, KM_NOSLEEP);
731         mutex_init(&kas.a_hat->hat_mutex, NULL, MUTEX_DEFAULT, NULL);
732         kas.a_hat->hat_as = &kas;
733         kas.a_hat->hat_flags = 0;
734         AS_LOCK_EXIT(&kas);
734         AS_LOCK_EXIT(&kas, &kas.a_lock);

736         CPUSET_ZERO(khat_cpuset);
737         CPUSET_ADD(khat_cpuset, CPU->cpu_id);

739         /*
740          * The kernel hat's next pointer serves as the head of the hat list .
741          * The kernel hat's prev pointer tracks the last hat on the list for
742          * htable_steal() to use.
743          */
744         kas.a_hat->hat_next = NULL;
745         kas.a_hat->hat_prev = NULL;

747         /*
748          * Allocate an htable hash bucket for the kernel
749          * XX64 - tune for 64 bit procs
750          */
751         kas.a_hat->hat_num_hash = mmu.hash_cnt;
752         kas.a_hat->hat_ht_hash = kmem_cache_alloc(hat_hash_cache, KM_NOSLEEP);
753         bzero(kas.a_hat->hat_ht_hash, mmu.hash_cnt * sizeof (htable_t *));

755         /*
756          * zero out the top level and cached htable pointers
757          */
758         kas.a_hat->hat_ht_cached = NULL;
759         kas.a_hat->hat_htable = NULL;
```

```
 761         /*
 762          * Pre-allocate hrm_hashtab before enabling the collection of
 763          * refmod statistics.  Allocating on the fly would mean us
 764          * running the risk of suffering recursive mutex enters or
 765          * deadlocks.
 766          */
 767         hrm_hashtab = kmem_zalloc(HRM_HASHSIZE * sizeof (struct hrmstat *),
 768             KM_SLEEP);
 769 }
_____unchanged_portion_omitted_

1137 /*
1138  * Unload all translations associated with an address space of a process
1139  * that is being swapped out.
1140  */
1141 void
1142 hat_swapout(hat_t *hat)
1143 {
1144         uintptr_t       vaddr = (uintptr_t)0;
1145         uintptr_t       eaddr = _userlimit;
1146         htable_t        *ht = NULL;
1147         level_t         l;

1149         XPV_DISALLOW_MIGRATE();
1150         /*
1151          * We can't just call hat_unload(hat, 0, _userlimit...)  here, because
1152          * seg_spt and shared pagetables can't be swapped out.
1153          * Take a look at segspt_shmswapout() - it's a big no-op.
1154          *
1155          * Instead we'll walk through all the address space and unload
1156          * any mappings which we are sure are not shared, not locked.
1157          */
1158         ASSERT(IS_PAGEALIGNED(vaddr));
1159         ASSERT(IS_PAGEALIGNED(eaddr));
1160         ASSERT(AS_LOCK_HELD(hat->hat_as));
1160         ASSERT(AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1161         if ((uintptr_t)hat->hat_as->a_userlimit < eaddr)
1162                 eaddr = (uintptr_t)hat->hat_as->a_userlimit;

1164         while (vaddr < eaddr) {
1165                 (void) htable_walk(hat, &ht, &vaddr, eaddr);
1166                 if (ht == NULL)
1167                         break;

1169                 ASSERT(!IN_VA_HOLE(vaddr));

1171                 /*
1172                  * If the page table is shared skip its entire range.
1173                  */
1174                 l = ht->ht_level;
1175                 if (ht->ht_flags & HTABLE_SHARED_PFN) {
1176                         vaddr = ht->ht_vaddr + LEVEL_SIZE(l + 1);
1177                         htable_release(ht);
1178                         ht = NULL;
1179                         continue;
1180                 }

1182                 /*
1183                  * If the page table has no locked entries, unload this one.
1184                  */
1185                 if (ht->ht_lock_cnt == 0)
1186                         hat_unload(hat, (caddr_t)vaddr, LEVEL_SIZE(l),
1187                             HAT_UNLOAD_UNMAP);

1189                 /*
1190                  * If we have a level 0 page table with locked entries,
```



```
1191                  * skip the entire page table, otherwise skip just one entry.
1192                  */
1193                 if (ht->ht_lock_cnt > 0 && l == 0)
1194                         vaddr = ht->ht_vaddr + LEVEL_SIZE(1);
1195                 else
1196                         vaddr += LEVEL_SIZE(l);
1197         }
1198         if (ht)
1199                 htable_release(ht);

1201         /*
1202          * We're in swapout because the system is low on memory, so
1203          * go back and flush all the htables off the cached list.
1204          */
1205         htable_purge_hat(hat);
1206         XPV_ALLOW_MIGRATE();
1207 }
_____unchanged_portion_omitted_

1415 /*
1416  * Internal routine to load a single page table entry. This only fails if
1417  * we attempt to overwrite a page table link with a large page.
1418  */
1419 static int
1420 hati_load_common(
1421         hat_t           *hat,
1422         uintptr_t       va,
1423         page_t          *pp,
1424         uint_t          attr,
1425         uint_t          flags,
1426         level_t         level,
1427         pfn_t           pfn)
1428 {
1429         htable_t        *ht;
1430         uint_t          entry;
1431         x86pte_t        pte;
1432         int             rv = 0;

1434         /*
1435          * The number 16 is arbitrary and here to catch a recursion problem
1436          * early before we blow out the kernel stack.
1437          */
1438         ++curthread->t_hatdepth;
1439         ASSERT(curthread->t_hatdepth < 16);

1441         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
1441         ASSERT(hat == kas.a_hat ||
1442             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));

1443         if (flags & HAT_LOAD_SHARE)
1444                 hat->hat_flags |= HAT_SHARED;

1446         /*
1447          * Find the page table that maps this page if it already exists.
1448          */
1449         ht = htable_lookup(hat, va, level);

1451         /*
1452          * We must have HAT_LOAD_NOCONSIST if page_t is NULL.
1453          */
1454         if (pp == NULL)
1455                 flags |= HAT_LOAD_NOCONSIST;

1457         if (ht == NULL) {
1458                 ht = htable_create(hat, va, level, NULL);
1459                 ASSERT(ht != NULL);
```

```
1460         }
1461         entry = htable_va2entry(va, ht);

1463         /*
1464          * a bunch of paranoid error checking
1465          */
1466         ASSERT(ht->ht_busy > 0);
1467         if (ht->ht_vaddr > va || va > HTABLE_LAST_PAGE(ht))
1468                 panic("hati_load_common: bad htable %p, va %p",
1469                     (void *)ht, (void *)va);
1470         ASSERT(ht->ht_level == level);

1472         /*
1473          * construct the new PTE
1474          */
1475         if (hat == kas.a_hat)
1476                 attr &= ~PROT_USER;
1477         pte = hati_mkpte(pfn, attr, level, flags);
1478         if (hat == kas.a_hat && va >= kernelbase)
1479                 PTE_SET(pte, mmu.pt_global);

1481         /*
1482          * establish the mapping
1483          */
1484         rv = hati_pte_map(ht, entry, pp, pte, flags, NULL);

1486         /*
1487          * release the htable and any reserves
1488          */
1489         htable_release(ht);
1490         --curthread->t_hatdepth;
1491         return (rv);
1492 }
```

*unchanged_portion_omitted_*

```
1536 /*
1537  * hat_memload() - load a translation to the given page struct
1538  *
1539  * Flags for hat_memload/hat_devload/hat_*attr.
1540  *
1541  *      HAT_LOAD        Default flags to load a translation to the page.
1542  *
1543  *      HAT_LOAD_LOCK   Lock down mapping resources; hat_map(), hat_memload(),
1544  *                      and hat_devload().
1545  *
1546  *      HAT_LOAD_NOCONSIST Do not add mapping to page_t mapping list.
1547  *                      sets PT_NOCONSIST
1548  *
1549  *      HAT_LOAD_SHARE  A flag to hat_memload() to indicate h/w page tables
1550  *                      that map some user pages (not kas) is shared by more
1551  *                      than one process (eg. ISM).
1552  *
1553  *      HAT_LOAD_REMAP  Reload a valid pte with a different page frame.
1554  *
1555  *      HAT_NO_KALLOC   Do not kmem_alloc while creating the mapping; at this
1556  *                      point, it's setting up mapping to allocate internal
1557  *                      hat layer data structures.  This flag forces hat layer
1558  *                      to tap its reserves in order to prevent infinite
1559  *                      recursion.
1560  *
1561  * The following is a protection attribute (like PROT_READ, etc.)
1562  *
1563  *      HAT_NOSYNC      set PT_NOSYNC - this mapping's ref/mod bits
1564  *                      are never cleared.
1565  *
1566  * Installing new valid PTE's and creation of the mapping list
```



```
1567  * entry are controlled under the same lock. It's derived from the
1568  * page_t being mapped.
1569  */
1570 static uint_t supported_memload_flags =
1571         HAT_LOAD | HAT_LOAD_LOCK | HAT_LOAD_ADV | HAT_LOAD_NOCONSIST |
1572         HAT_LOAD_SHARE | HAT_NO_KALLOC | HAT_LOAD_REMAP | HAT_LOAD_TEXT;

1574 void
1575 hat_memload(
1576         hat_t           *hat,
1577         caddr_t         addr,
1578         page_t          *pp,
1579         uint_t          attr,
1580         uint_t          flags)
1581 {
1582         uintptr_t       va = (uintptr_t)addr;
1583         level_t         level = 0;
1584         pfn_t           pfn = page_pptonum(pp);

1586         XPV_DISALLOW_MIGRATE();
1587         ASSERT(IS_PAGEALIGNED(va));
1588         ASSERT(hat == kas.a_hat || va < _userlimit);
1589         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
1590         ASSERT(hat == kas.a_hat ||
1591             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1590         ASSERT((flags & supported_memload_flags) == flags);

1592         ASSERT(!IN_VA_HOLE(va));
1593         ASSERT(!PP_ISFREE(pp));

1595         /*
1596          * kernel address special case for performance.
1597          */
1598         if (mmu.kmap_addr <= va && va < mmu.kmap_eaddr) {
1599                 ASSERT(hat == kas.a_hat);
1600                 hat_kmap_load(addr, pp, attr, flags);
1601                 XPV_ALLOW_MIGRATE();
1602                 return;
1603         }

1605         /*
1606          * This is used for memory with normal caching enabled, so
1607          * always set HAT_STORECACHING_OK.
1608          */
1609         attr |= HAT_STORECACHING_OK;
1610         if (hati_load_common(hat, va, pp, attr, flags, level, pfn) != 0)
1611                 panic("unexpected hati_load_common() failure");
1612         XPV_ALLOW_MIGRATE();
1613 }
```

*unchanged_portion_omitted_*

```
1623 /*
1624  * Load the given array of page structs using large pages when possible
1625  */
1626 void
1627 hat_memload_array(
1628         hat_t           *hat,
1629         caddr_t         addr,
1630         size_t          len,
1631         page_t          **pages,
1632         uint_t          attr,
1633         uint_t          flags)
1634 {
1635         uintptr_t       va = (uintptr_t)addr;
1636         uintptr_t       eaddr = va + len;
1637         level_t         level;
```

```
1638         size_t          pgsize;
1639         pgcnt_t         pgindx = 0;
1640         pfn_t           pfn;
1641         pgcnt_t         i;

1643         XPV_DISALLOW_MIGRATE();
1644         ASSERT(IS_PAGEALIGNED(va));
1645         ASSERT(hat == kas.a_hat || va + len <= _userlimit);
1646         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
1648         ASSERT(hat == kas.a_hat ||
1649             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1647         ASSERT((flags & supported_memload_flags) == flags);

1649         /*
1650          * memload is used for memory with full caching enabled, so
1651          * set HAT_STORECACHING_OK.
1652          */
1653         attr |= HAT_STORECACHING_OK;

1655         /*
1656          * handle all pages using largest possible pagesize
1657          */
1658         while (va < eaddr) {
1659                 /*
1660                  * decide what level mapping to use (ie. pagesize)
1661                  */
1662                 pfn = page_pptonum(pages[pgindx]);
1663                 for (level = mmu.max_page_level; ; --level) {
1664                         pgsize = LEVEL_SIZE(level);
1665                         if (level == 0)
1666                                 break;

1668                         if (!IS_P2ALIGNED(va, pgsize) ||
1669                             (eaddr - va) < pgsize ||
1670                             !IS_P2ALIGNED(pfn_to_pa(pfn), pgsize))
1671                                 continue;

1673                         /*
1674                          * To use a large mapping of this size, all the
1675                          * pages we are passed must be sequential subpages
1676                          * of the large page.
1677                          * hat_page_demote() can't change p_szc because
1678                          * all pages are locked.
1679                          */
1680                         if (pages[pgindx]->p_szc >= level) {
1681                                 for (i = 0; i < mmu_btop(pgsize); ++i) {
1682                                         if (pfn + i !=
1683                                             page_pptonum(pages[pgindx + i]))
1684                                                 break;
1685                                         ASSERT(pages[pgindx + i]->p_szc >=
1686                                             level);
1687                                         ASSERT(pages[pgindx] + i ==
1688                                             pages[pgindx + i]);
1689                                 }
1690                                 if (i == mmu_btop(pgsize)) {
1691 #ifdef DEBUG
1692                                         if (level == 2)
1693                                                 map1gcnt++;
1694 #endif
1695                                         break;
1696                                 }
1697                         }
1698                 }

1700                 /*
1701                  * Load this page mapping. If the load fails, try a smaller
```



```
1702                  * pagesize.
1703                  */
1704                 ASSERT(!IN_VA_HOLE(va));
1705                 while (hati_load_common(hat, va, pages[pgindx], attr,
1706                     flags, level, pfn) != 0) {
1707                         if (level == 0)
1708                                 panic("unexpected hati_load_common() failure");
1709                         --level;
1710                         pgsize = LEVEL_SIZE(level);
1711                 }

1713                 /*
1714                  * move to next page
1715                  */
1716                 va += pgsize;
1717                 pgindx += mmu_btop(pgsize);
1718         }
1719         XPV_ALLOW_MIGRATE();
1720 }
_____unchanged_portion_omitted_

1731 /*
1732  * void hat_devload(hat, addr, len, pf, attr, flags)
1733  *      load/lock the given page frame number
1734  *
1735  * Advisory ordering attributes. Apply only to device mappings.
1736  *
1737  * HAT_STRICTORDER: the CPU must issue the references in order, as the
1738  *      programmer specified.  This is the default.
1739  * HAT_UNORDERED_OK: the CPU may reorder the references (this is all kinds
1740  *      of reordering; store or load with store or load).
1741  * HAT_MERGING_OK: merging and batching: the CPU may merge individual stores
1742  *      to consecutive locations (for example, turn two consecutive byte
1743  *      stores into one halfword store), and it may batch individual loads
1744  *      (for example, turn two consecutive byte loads into one halfword load).
1745  *      This also implies re-ordering.
1746  * HAT_LOADCACHING_OK: the CPU may cache the data it fetches and reuse it
1747  *      until another store occurs.  The default is to fetch new data
1748  *      on every load.  This also implies merging.
1749  * HAT_STORECACHING_OK: the CPU may keep the data in the cache and push it to
1750  *      the device (perhaps with other data) at a later time.  The default is
1751  *      to push the data right away.  This also implies load caching.
1752  *
1753  * Equivalent of hat_memload(), but can be used for device memory where
1754  * there are no page_t's and we support additional flags (write merging, etc).
1755  * Note that we can have large page mappings with this interface.
1756  */
1757 int supported_devload_flags = HAT_LOAD | HAT_LOAD_LOCK |
1758         HAT_LOAD_NOCONSIST | HAT_STRICTORDER | HAT_UNORDERED_OK |
1759         HAT_MERGING_OK | HAT_LOADCACHING_OK | HAT_STORECACHING_OK;

1761 void
1762 hat_devload(
1763         hat_t           *hat,
1764         caddr_t         addr,
1765         size_t          len,
1766         pfn_t           pfn,
1767         uint_t          attr,
1768         int             flags)
1769 {
1770         uintptr_t       va = ALIGN2PAGE(addr);
1771         uintptr_t       eva = va + len;
1772         level_t         level;
1773         size_t          pgsize;
1774         page_t          *pp;
1775         int             f;      /* per PTE copy of flags  - maybe modified */
```

```
1776         uint_t          a;      /* per PTE copy of attr */

1778         XPV_DISALLOW_MIGRATE();
1779         ASSERT(IS_PAGEALIGNED(va));
1780         ASSERT(hat == kas.a_hat || eva <= _userlimit);
1781         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
1784         ASSERT(hat == kas.a_hat ||
1785             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1782         ASSERT((flags & supported_devload_flags) == flags);

1784         /*
1785          * handle all pages
1786          */
1787         while (va < eva) {

1789                 /*
1790                  * decide what level mapping to use (ie. pagesize)
1791                  */
1792                 for (level = mmu.max_page_level; ; --level) {
1793                         pgsize = LEVEL_SIZE(level);
1794                         if (level == 0)
1795                                 break;
1796                         if (IS_P2ALIGNED(va, pgsize) &&
1797                             (eva - va) >= pgsize &&
1798                             IS_P2ALIGNED(pfn, mmu_btop(pgsize))) {
1799 #ifdef DEBUG
1800                                 if (level == 2)
1801                                         map1gcnt++;
1802 #endif
1803                                 break;
1804                         }
1805                 }

1807                 /*
1808                  * If this is just memory then allow caching (this happens
1809                  * for the nucleus pages) - though HAT_PLAT_NOCACHE can be used
1810                  * to override that. If we don't have a page_t then make sure
1811                  * NOCONSIST is set.
1812                  */
1813                 a = attr;
1814                 f = flags;
1815                 if (!pf_is_memory(pfn))
1816                         f |= HAT_LOAD_NOCONSIST;
1817                 else if (!(a & HAT_PLAT_NOCACHE))
1818                         a |= HAT_STORECACHING_OK;

1820                 if (f & HAT_LOAD_NOCONSIST)
1821                         pp = NULL;
1822                 else
1823                         pp = page_numtopp_nolock(pfn);

1825                 /*
1826                  * Check to make sure we are really trying to map a valid
1827                  * memory page. The caller wishing to intentionally map
1828                  * free memory pages will have passed the HAT_LOAD_NOCONSIST
1829                  * flag, then pp will be NULL.
1830                  */
1831                 if (pp != NULL) {
1832                         if (PP_ISFREE(pp)) {
1833                                 panic("hat_devload: loading "
1834                                     "a mapping to free page %p", (void *)pp);
1835                         }

1837                         if (!PAGE_LOCKED(pp) && !PP_ISNORELOC(pp)) {
1838                                 panic("hat_devload: loading a mapping "
1839                                     "to an unlocked page %p",
```



```
1840                                     (void *)pp);
1841                         }
1842                 }

1844                 /*
1845                  * load this page mapping
1846                  */
1847                 ASSERT(!IN_VA_HOLE(va));
1848                 while (hati_load_common(hat, va, pp, a, f, level, pfn) != 0) {
1849                         if (level == 0)
1850                                 panic("unexpected hati_load_common() failure");
1851                         --level;
1852                         pgsize = LEVEL_SIZE(level);
1853                 }

1855                 /*
1856                  * move to next page
1857                  */
1858                 va += pgsize;
1859                 pfn += mmu_btop(pgsize);
1860         }
1861         XPV_ALLOW_MIGRATE();
1862 }

1864 /*
1865  * void hat_unlock(hat, addr, len)
1866  *      unlock the mappings to a given range of addresses
1867  *
1868  * Locks are tracked by ht_lock_cnt in the htable.
1869  */
1870 void
1871 hat_unlock(hat_t *hat, caddr_t addr, size_t len)
1872 {
1873         uintptr_t       vaddr = (uintptr_t)addr;
1874         uintptr_t       eaddr = vaddr + len;
1875         htable_t        *ht = NULL;

1877         /*
1878          * kernel entries are always locked, we don't track lock counts
1879          */
1880         ASSERT(hat == kas.a_hat || eaddr <= _userlimit);
1881         ASSERT(IS_PAGEALIGNED(vaddr));
1882         ASSERT(IS_PAGEALIGNED(eaddr));
1883         if (hat == kas.a_hat)
1884                 return;
1885         if (eaddr > _userlimit)
1886                 panic("hat_unlock() address out of range - above _userlimit");

1888         XPV_DISALLOW_MIGRATE();
1889         ASSERT(AS_LOCK_HELD(hat->hat_as));
1893         ASSERT(AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1890         while (vaddr < eaddr) {
1891                 (void) htable_walk(hat, &ht, &vaddr, eaddr);
1892                 if (ht == NULL)
1893                         break;

1895                 ASSERT(!IN_VA_HOLE(vaddr));

1897                 if (ht->ht_lock_cnt < 1)
1898                         panic("hat_unlock(): lock_cnt < 1, "
1899                             "htable=%p, vaddr=%p\n", (void *)ht, (void *)vaddr);
1900                 HTABLE_LOCK_DEC(ht);

1902                 vaddr += LEVEL_SIZE(ht->ht_level);
1903         }
1904         if (ht)
```

```
1905                htable_release(ht);
1906        XPV_ALLOW_MIGRATE();
1907 }
_____unchanged_portion_omitted_

2624 /*
2625  * hat_updateattr() applies the given attribute change to an existing mapping
2626  */
2627 #define HAT_LOAD_ATTR           1
2628 #define HAT_SET_ATTR            2
2629 #define HAT_CLR_ATTR            3

2631 static void
2632 hat_updateattr(hat_t *hat, caddr_t addr, size_t len, uint_t attr, int what)
2633 {
2634         uintptr_t       vaddr = (uintptr_t)addr;
2635         uintptr_t       eaddr = (uintptr_t)addr + len;
2636         htable_t        *ht = NULL;
2637         uint_t          entry;
2638         x86pte_t        oldpte, newpte;
2639         page_t          *pp;

2641         XPV_DISALLOW_MIGRATE();
2642         ASSERT(IS_PAGEALIGNED(vaddr));
2643         ASSERT(IS_PAGEALIGNED(eaddr));
2644         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
2648         ASSERT(hat == kas.a_hat ||
2649             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
2645         for (; vaddr < eaddr; vaddr += LEVEL_SIZE(ht->ht_level)) {
2646 try_again:
2647                 oldpte = htable_walk(hat, &ht, &vaddr, eaddr);
2648                 if (ht == NULL)
2649                         break;
2650                 if (PTE_GET(oldpte, PT_SOFTWARE) >= PT_NOCONSIST)
2651                         continue;

2653                 pp = page_numtopp_nolock(PTE2PFN(oldpte, ht->ht_level));
2654                 if (pp == NULL)
2655                         continue;
2656                 x86_hm_enter(pp);

2658                 newpte = oldpte;
2659                 /*
2660                  * We found a page table entry in the desired range,
2661                  * figure out the new attributes.
2662                  */
2663                 if (what == HAT_SET_ATTR || what == HAT_LOAD_ATTR) {
2664                         if ((attr & PROT_WRITE) &&
2665                             !PTE_GET(oldpte, PT_WRITABLE))
2666                                 newpte |= PT_WRITABLE;

2668                         if ((attr & HAT_NOSYNC) &&
2669                             PTE_GET(oldpte, PT_SOFTWARE) < PT_NOSYNC)
2670                                 newpte |= PT_NOSYNC;

2672                         if ((attr & PROT_EXEC) && PTE_GET(oldpte, mmu.pt_nx))
2673                                 newpte &= ~mmu.pt_nx;
2674                 }

2676                 if (what == HAT_LOAD_ATTR) {
2677                         if (!(attr & PROT_WRITE) &&
2678                             PTE_GET(oldpte, PT_WRITABLE))
2679                                 newpte &= ~PT_WRITABLE;

2681                         if (!(attr & HAT_NOSYNC) &&
2682                             PTE_GET(oldpte, PT_SOFTWARE) >= PT_NOSYNC)
```



```
2683                                 newpte &= ~PT_SOFTWARE;

2685                         if (!(attr & PROT_EXEC) && !PTE_GET(oldpte, mmu.pt_nx))
2686                                 newpte |= mmu.pt_nx;
2687                 }

2689                 if (what == HAT_CLR_ATTR) {
2690                         if ((attr & PROT_WRITE) && PTE_GET(oldpte, PT_WRITABLE))
2691                                 newpte &= ~PT_WRITABLE;

2693                         if ((attr & HAT_NOSYNC) &&
2694                             PTE_GET(oldpte, PT_SOFTWARE) >= PT_NOSYNC)
2695                                 newpte &= ~PT_SOFTWARE;

2697                         if ((attr & PROT_EXEC) && !PTE_GET(oldpte, mmu.pt_nx))
2698                                 newpte |= mmu.pt_nx;
2699                 }

2701                 /*
2702                  * Ensure NOSYNC/NOCONSIST mappings have REF and MOD set.
2703                  * x86pte_set() depends on this.
2704                  */
2705                 if (PTE_GET(newpte, PT_SOFTWARE) >= PT_NOSYNC)
2706                         newpte |= PT_REF | PT_MOD;

2708                 /*
2709                  * what about PROT_READ or others? this code only handles:
2710                  * EXEC, WRITE, NOSYNC
2711                  */

2713                 /*
2714                  * If new PTE really changed, update the table.
2715                  */
2716                 if (newpte != oldpte) {
2717                         entry = htable_va2entry(vaddr, ht);
2718                         oldpte = hati_update_pte(ht, entry, oldpte, newpte);
2719                         if (oldpte != 0) {
2720                                 x86_hm_exit(pp);
2721                                 goto try_again;
2722                         }
2723                 }
2724                 x86_hm_exit(pp);
2725         }
2726         if (ht)
2727                 htable_release(ht);
2728         XPV_ALLOW_MIGRATE();
2729 }
_____unchanged_portion_omitted_

2840 /*
2841  * int hat_probe(hat, addr)
2842  *      return 0 if no valid mapping is present.  Faster version
2843  *      of hat_getattr in certain architectures.
2844  */
2845 int
2846 hat_probe(hat_t *hat, caddr_t addr)
2847 {
2848         uintptr_t       vaddr = ALIGN2PAGE(addr);
2849         uint_t          entry;
2850         htable_t        *ht;
2851         pgcnt_t         pg_off;

2853         ASSERT(hat == kas.a_hat || vaddr <= _userlimit);
2854         ASSERT(hat == kas.a_hat || AS_LOCK_HELD(hat->hat_as));
2859         ASSERT(hat == kas.a_hat ||
2860             AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
```

```
2855         if (IN_VA_HOLE(vaddr))
2856                 return (0);

2858         /*
2859          * Most common use of hat_probe is from segmap. We special case it
2860          * for performance.
2861          */
2862         if (mmu.kmap_addr <= vaddr && vaddr < mmu.kmap_eaddr) {
2863                 pg_off = mmu_btop(vaddr - mmu.kmap_addr);
2864                 if (mmu.pae_hat)
2865                         return (PTE_ISVALID(mmu.kmap_ptes[pg_off]));
2866                 else
2867                         return (PTE_ISVALID(
2868                             ((x86pte32_t *)mmu.kmap_ptes)[pg_off]));
2869         }

2871         ht = htable_getpage(hat, vaddr, &entry);
2872         htable_release(ht);
2873         return (ht != NULL);
2874 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8755 Wed Nov 25 13:59:42 2015
new/usr/src/uts/i86xpv/io/privcmd.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 112 /*
 113  * Map a contiguous set of machine frames in a foreign domain.
 114  * Used in the following way:
 115  *
 116  *      privcmd_mmap_t p;
 117  *      privcmd_mmap_entry_t e;
 118  *
 119  *      addr = mmap(NULL, size, prot, MAP_SHARED, fd, 0);
 120  *      p.num = number of privcmd_mmap_entry_t's
 121  *      p.dom = domid;
 122  *      p.entry = &e;
 123  *      e.va = addr;
 124  *      e.mfn = mfn;
 125  *      e.npages = btopr(size);
 126  *      ioctl(fd, IOCTL_PRIVCMD_MMAP, &p);
 127  */
 128 /*ARGSUSED2*/
 129 int
 130 do_privcmd_mmap(void *uarg, int mode, cred_t *cr)
 131 {
 132         privcmd_mmap_t __mmapcmd, *mmc = &__mmapcmd;
 133         privcmd_mmap_entry_t *umme;
 134         struct as *as = curproc->p_as;
 135         struct seg *seg;
 136         int i, error = 0;

 138         if (ddi_copyin(uarg, mmc, sizeof (*mmc), mode))
 139                 return (EFAULT);

 141         DTRACE_XPV3(mmap__start, domid_t, mmc->dom, int, mmc->num,
 142             privcmd_mmap_entry_t *, mmc->entry);

 144         if (mmc->dom == DOMID_SELF) {
 145                 error = ENOTSUP;        /* Too paranoid? */
 146                 goto done;
 147         }

 149         for (umme = mmc->entry, i = 0; i < mmc->num; i++, umme++) {
 150                 privcmd_mmap_entry_t __mmapent, *mme = &__mmapent;
 151                 caddr_t addr;

 153                 if (ddi_copyin(umme, mme, sizeof (*mme), mode)) {
 154                         error = EFAULT;
 155                         break;
 156                 }

 158                 DTRACE_XPV3(mmap__entry, ulong_t, mme->va, ulong_t, mme->mfn,
 159                     ulong_t, mme->npages);

 161                 if (mme->mfn == MFN_INVALID) {
 162                         error = EINVAL;
 163                         break;
 164                 }

 166                 addr = (caddr_t)mme->va;

 168                 /*
 169                  * Find the segment we want to mess with, then add
 170                  * the mfn range to the segment.
```



```
 171                  */
 172                 AS_LOCK_ENTER(as, RW_READER);
 172                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 173                 if ((seg = as_findseg(as, addr, 0)) == NULL ||
 174                     addr + mmu_ptob(mme->npages) > seg->s_base + seg->s_size)
 175                         error = EINVAL;
 176                 else
 177                         error = segmf_add_mfns(seg, addr,
 178                             mme->mfn, mme->npages, mmc->dom);
 179                 AS_LOCK_EXIT(as);
 179                 AS_LOCK_EXIT(as, &as->a_lock);

 181                 if (error != 0)
 182                         break;
 183         }

 185 done:
 186         DTRACE_XPV1(mmap__end, int, error);

 188         return (error);
 189 }

 191 /*
 192  * Set up the address range to map to an array of mfns in
 193  * a foreign domain.  Used in the following way:
 194  *
 195  *      privcmd_mmap_batch_t p;
 196  *
 197  *      addr = mmap(NULL, size, prot, MAP_SHARED, fd, 0);
 198  *      p.num = number of pages
 199  *      p.dom = domid
 200  *      p.addr = addr;
 201  *      p.arr = array of mfns, indexed 0 .. p.num - 1
 202  *      ioctl(fd, IOCTL_PRIVCMD_MMAPBATCH, &p);
 203  */
 204 /*ARGSUSED2*/
 205 static int
 206 do_privcmd_mmapbatch(void *uarg, int mode, cred_t *cr)
 207 {
 208         privcmd_mmapbatch_t __mmapbatch, *mmb = &__mmapbatch;
 209         struct as *as = curproc->p_as;
 210         struct seg *seg;
 211         int i, error = 0;
 212         caddr_t addr;
 213         ulong_t *ulp;

 215         if (ddi_copyin(uarg, mmb, sizeof (*mmb), mode))
 216                 return (EFAULT);

 218         DTRACE_XPV3(mmapbatch__start, domid_t, mmb->dom, int, mmb->num,
 219             caddr_t, mmb->addr);

 221         addr = (caddr_t)mmb->addr;
 222         AS_LOCK_ENTER(as, RW_READER);
 222         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 223         if ((seg = as_findseg(as, addr, 0)) == NULL ||
 224             addr + ptob(mmb->num) > seg->s_base + seg->s_size) {
 225                 error = EINVAL;
 226                 goto done;
 227         }

 229         for (i = 0, ulp = mmb->arr;
 230             i < mmb->num; i++, addr += PAGESIZE, ulp++) {
 231                 mfn_t mfn;

 233                 if (fulword(ulp, &mfn) != 0) {
```

```
234                         error = EFAULT;
235                         break;
236                 }

238                 if (mfn == MFN_INVALID) {
239                         /*
240                          * This mfn is invalid and should not be added to
241                          * segmf, as we'd only cause an immediate EFAULT when
242                          * we tried to fault it in.
243                          */
244                         mfn |= XEN_DOMCTL_PFINFO_XTAB;
245                         continue;
246                 }

248                 if (segmf_add_mfns(seg, addr, mfn, 1, mmb->dom) == 0)
249                         continue;

251                 /*
252                  * Tell the process that this MFN could not be mapped, so it
253                  * won't later try to access it.
254                  */
255                 mfn |= XEN_DOMCTL_PFINFO_XTAB;
256                 if (sulword(ulp, mfn) != 0) {
257                         error = EFAULT;
258                         break;
259                 }
260         }

262 done:
263         AS_LOCK_EXIT(as);
263         AS_LOCK_EXIT(as, &as->a_lock);

265         DTRACE_XPV3(mmapbatch__end, int, error, struct seg *, seg, caddr_t,
266             mmb->addr);

268         return (error);
269 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
  422250 Wed Nov 25 13:59:42 2015
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

1054 /*
1055  * Initialize the hardware address translation structures.
1056  */
1057 void
1058 hat_init(void)
1059 {
1060         int             i;
1061         uint_t          sz;
1062         size_t          size;

1064         hat_lock_init();
1065         hat_kstat_init();

1067         /*
1068          * Hardware-only bits in a TTE
1069          */
1070         MAKE_TTE_MASK(&hw_tte);

1072         hat_init_pagesizes();

1074         /* Initialize the hash locks */
1075         for (i = 0; i < khmehash_num; i++) {
1076                 mutex_init(&khme_hash[i].hmehash_mutex, NULL,
1077                     MUTEX_DEFAULT, NULL);
1078                 khme_hash[i].hmeh_nextpa = HMEBLK_ENDPA;
1079         }
1080         for (i = 0; i < uhmehash_num; i++) {
1081                 mutex_init(&uhme_hash[i].hmehash_mutex, NULL,
1082                     MUTEX_DEFAULT, NULL);
1083                 uhme_hash[i].hmeh_nextpa = HMEBLK_ENDPA;
1084         }
1085         khmehash_num--;         /* make sure counter starts from 0 */
1086         uhmehash_num--;         /* make sure counter starts from 0 */

1088         /*
1089          * Allocate context domain structures.
1090          *
1091          * A platform may choose to modify max_mmu_ctxdoms in
1092          * set_platform_defaults(). If a platform does not define
1093          * a set_platform_defaults() or does not choose to modify
1094          * max_mmu_ctxdoms, it gets one MMU context domain for every CPU.
1095          *
1096          * For all platforms that have CPUs sharing MMUs, this
1097          * value must be defined.
1098          */
1099         if (max_mmu_ctxdoms == 0)
1100                 max_mmu_ctxdoms = max_ncpus;

1102         size = max_mmu_ctxdoms * sizeof (mmu_ctx_t *);
1103         mmu_ctxs_tbl = kmem_zalloc(size, KM_SLEEP);

1105         /* mmu_ctx_t is 64 bytes aligned */
1106         mmuctxdom_cache = kmem_cache_create("mmuctxdom_cache",
1107             sizeof (mmu_ctx_t), 64, NULL, NULL, NULL, NULL, 0);
1108         /*
1109          * MMU context domain initialization for the Boot CPU.
1110          * This needs the context domains array allocated above.
1111          */
1112         mutex_enter(&cpu_lock);
```



```
1113         sfmmu_cpu_init(CPU);
1114         mutex_exit(&cpu_lock);

1116         /*
1117          * Intialize ism mapping list lock.
1118          */

1120         mutex_init(&ism_mlist_lock, NULL, MUTEX_DEFAULT, NULL);

1122         /*
1123          * Each sfmmu structure carries an array of MMU context info
1124          * structures, one per context domain. The size of this array depends
1125          * on the maximum number of context domains. So, the size of the
1126          * sfmmu structure varies per platform.
1127          *
1128          * sfmmu is allocated from static arena, because trap
1129          * handler at TL > 0 is not allowed to touch kernel relocatable
1130          * memory. sfmmu's alignment is changed to 64 bytes from
1131          * default 8 bytes, as the lower 6 bits will be used to pass
1132          * pgcnt to vtag_flush_pgcnt_tl1.
1133          */
1134         size = sizeof (sfmmu_t) + sizeof (sfmmu_ctx_t) * (max_mmu_ctxdoms - 1);

1136         sfmmuid_cache = kmem_cache_create("sfmmuid_cache", size,
1137             64, sfmmu_idcache_constructor, sfmmu_idcache_destructor,
1138             NULL, NULL, static_arena, 0);

1140         sfmmu_tsbinfo_cache = kmem_cache_create("sfmmu_tsbinfo_cache",
1141             sizeof (struct tsb_info), 0, NULL, NULL, NULL, NULL, NULL, 0);

1143         /*
1144          * Since we only use the tsb8k cache to "borrow" pages for TSBs
1145          * from the heap when low on memory or when TSB_FORCEALLOC is
1146          * specified, don't use magazines to cache them--we want to return
1147          * them to the system as quickly as possible.
1148          */
1149         sfmmu_tsb8k_cache = kmem_cache_create("sfmmu_tsb8k_cache",
1150             MMU_PAGESIZE, MMU_PAGESIZE, NULL, NULL, NULL, NULL,
1151             static_arena, KMC_NOMAGAZINE);

1153         /*
1154          * Set tsb_alloc_hiwater to 1/tsb_alloc_hiwater_factor of physical
1155          * memory, which corresponds to the old static reserve for TSBs.
1156          * tsb_alloc_hiwater_factor defaults to 32.  This caps the amount of
1157          * memory we'll allocate for TSB slabs; beyond this point TSB
1158          * allocations will be taken from the kernel heap (via
1159          * sfmmu_tsb8k_cache) and will be throttled as would any other kmem
1160          * consumer.
1161          */
1162         if (tsb_alloc_hiwater_factor == 0) {
1163                 tsb_alloc_hiwater_factor = TSB_ALLOC_HIWATER_FACTOR_DEFAULT;
1164         }
1165         SFMMU_SET_TSB_ALLOC_HIWATER(physmem);

1167         for (sz = tsb_slab_ttesz; sz > 0; sz--) {
1168                 if (!(disable_large_pages & (1 << sz)))
1169                         break;
1170         }

1172         if (sz < tsb_slab_ttesz) {
1173                 tsb_slab_ttesz = sz;
1174                 tsb_slab_shift = MMU_PAGESHIFT + (sz << 1) + sz;
1175                 tsb_slab_size = 1 << tsb_slab_shift;
1176                 tsb_slab_mask = (1 << (tsb_slab_shift - MMU_PAGESHIFT)) - 1;
1177                 use_bigtsb_arena = 0;
1178         } else if (use_bigtsb_arena &&
```

```
1179             (disable_large_pages & (1 << bigtsb_slab_ttesz))) {
1180                 use_bigtsb_arena = 0;
1181         }

1183         if (!use_bigtsb_arena) {
1184                 bigtsb_slab_shift = tsb_slab_shift;
1185         }
1186         SFMMU_SET_TSB_MAX_GROWSIZE(physmem);

1188         /*
1189          * On smaller memory systems, allocate TSB memory in smaller chunks
1190          * than the default 4M slab size. We also honor disable_large_pages
1191          * here.
1192          *
1193          * The trap handlers need to be patched with the final slab shift,
1194          * since they need to be able to construct the TSB pointer at runtime.
1195          */
1196         if ((tsb_max_growsize <= TSB_512K_SZCODE) &&
1197             !(disable_large_pages & (1 << TTE512K))) {
1198                 tsb_slab_ttesz = TTE512K;
1199                 tsb_slab_shift = MMU_PAGESHIFT512K;
1200                 tsb_slab_size = MMU_PAGESIZE512K;
1201                 tsb_slab_mask = MMU_PAGEOFFSET512K >> MMU_PAGESHIFT;
1202                 use_bigtsb_arena = 0;
1203         }

1205         if (!use_bigtsb_arena) {
1206                 bigtsb_slab_ttesz = tsb_slab_ttesz;
1207                 bigtsb_slab_shift = tsb_slab_shift;
1208                 bigtsb_slab_size = tsb_slab_size;
1209                 bigtsb_slab_mask = tsb_slab_mask;
1210         }


1213         /*
1214          * Set up memory callback to update tsb_alloc_hiwater and
1215          * tsb_max_growsize.
1216          */
1217         i = kphysm_setup_func_register(&sfmmu_update_vec, (void *) 0);
1218         ASSERT(i == 0);

1220         /*
1221          * kmem_tsb_arena is the source from which large TSB slabs are
1222          * drawn.  The quantum of this arena corresponds to the largest
1223          * TSB size we can dynamically allocate for user processes.
1224          * Currently it must also be a supported page size since we
1225          * use exactly one translation entry to map each slab page.
1226          *
1227          * The per-lgroup kmem_tsb_default_arena arenas are the arenas from
1228          * which most TSBs are allocated.  Since most TSB allocations are
1229          * typically 8K we have a kmem cache we stack on top of each
1230          * kmem_tsb_default_arena to speed up those allocations.
1231          *
1232          * Note the two-level scheme of arenas is required only
1233          * because vmem_create doesn't allow us to specify alignment
1234          * requirements.  If this ever changes the code could be
1235          * simplified to use only one level of arenas.
1236          *
1237          * If 256M page support exists on sun4v, 256MB kmem_bigtsb_arena
1238          * will be provided in addition to the 4M kmem_tsb_arena.
1239          */
1240         if (use_bigtsb_arena) {
1241                 kmem_bigtsb_arena = vmem_create("kmem_bigtsb", NULL, 0,
1242                     bigtsb_slab_size, sfmmu_vmem_xalloc_aligned_wrapper,
1243                     vmem_xfree, heap_arena, 0, VM_SLEEP);
1244         }
```



```
1246         kmem_tsb_arena = vmem_create("kmem_tsb", NULL, 0, tsb_slab_size,
1247             sfmmu_vmem_xalloc_aligned_wrapper,
1248             vmem_xfree, heap_arena, 0, VM_SLEEP);

1250         if (tsb_lgrp_affinity) {
1251                 char s[50];
1252                 for (i = 0; i < NLGRPS_MAX; i++) {
1253                         if (use_bigtsb_arena) {
1254                                 (void) sprintf(s, "kmem_bigtsb_lgrp%d", i);
1255                                 kmem_bigtsb_default_arena[i] = vmem_create(s,
1256                                     NULL, 0, 2 * tsb_slab_size,
1257                                     sfmmu_tsb_segkmem_alloc,
1258                                     sfmmu_tsb_segkmem_free, kmem_bigtsb_arena,
1259                                     0, VM_SLEEP | VM_BESTFIT);
1260                         }

1262                         (void) sprintf(s, "kmem_tsb_lgrp%d", i);
1263                         kmem_tsb_default_arena[i] = vmem_create(s,
1264                             NULL, 0, PAGESIZE, sfmmu_tsb_segkmem_alloc,
1265                             sfmmu_tsb_segkmem_free, kmem_tsb_arena, 0,
1266                             VM_SLEEP | VM_BESTFIT);

1268                         (void) sprintf(s, "sfmmu_tsb_lgrp%d_cache", i);
1269                         sfmmu_tsb_cache[i] = kmem_cache_create(s,
1270                             PAGESIZE, PAGESIZE, NULL, NULL, NULL, NULL,
1271                             kmem_tsb_default_arena[i], 0);
1272                 }
1273         } else {
1274                 if (use_bigtsb_arena) {
1275                         kmem_bigtsb_default_arena[0] =
1276                             vmem_create("kmem_bigtsb_default", NULL, 0,
1277                             2 * tsb_slab_size, sfmmu_tsb_segkmem_alloc,
1278                             sfmmu_tsb_segkmem_free, kmem_bigtsb_arena, 0,
1279                             VM_SLEEP | VM_BESTFIT);
1280                 }

1282                 kmem_tsb_default_arena[0] = vmem_create("kmem_tsb_default",
1283                     NULL, 0, PAGESIZE, sfmmu_tsb_segkmem_alloc,
1284                     sfmmu_tsb_segkmem_free, kmem_tsb_arena, 0,
1285                     VM_SLEEP | VM_BESTFIT);
1286                 sfmmu_tsb_cache[0] = kmem_cache_create("sfmmu_tsb_cache",
1287                     PAGESIZE, PAGESIZE, NULL, NULL, NULL, NULL,
1288                     kmem_tsb_default_arena[0], 0);
1289         }

1291         sfmmu8_cache = kmem_cache_create("sfmmu8_cache", HME8BLK_SZ,
1292             HMEBLK_ALIGN, sfmmu_hblkcache_constructor,
1293             sfmmu_hblkcache_destructor,
1294             sfmmu_hblkcache_reclaim, (void *)HME8BLK_SZ,
1295             hat_memload_arena, KMC_NOHASH);

1297         hat_memload1_arena = vmem_create("hat_memload1", NULL, 0, PAGESIZE,
1298             segkmem_alloc_permanent, segkmem_free, heap_arena, 0,
1299             VMC_DUMPSAFE | VM_SLEEP);

1301         sfmmu1_cache = kmem_cache_create("sfmmu1_cache", HME1BLK_SZ,
1302             HMEBLK_ALIGN, sfmmu_hblkcache_constructor,
1303             sfmmu_hblkcache_destructor,
1304             NULL, (void *)HME1BLK_SZ,
1305             hat_memload1_arena, KMC_NOHASH);

1307         pa_hment_cache = kmem_cache_create("pa_hment_cache", PAHME_SZ,
1308             0, NULL, NULL, NULL, NULL, static_arena, KMC_NOHASH);

1310         ism_blk_cache = kmem_cache_create("ism_blk_cache",
```

```
1311             sizeof (ism_blk_t), ecache_alignsize, NULL, NULL,
1312             NULL, NULL, static_arena, KMC_NOHASH);

1314         ism_ment_cache = kmem_cache_create("ism_ment_cache",
1315             sizeof (ism_ment_t), 0, NULL, NULL,
1316             NULL, NULL, NULL, 0);

1318         /*
1319          * We grab the first hat for the kernel,
1320          */
1321         AS_LOCK_ENTER(&kas, RW_WRITER);
1321         AS_LOCK_ENTER(&kas, &kas.a_lock, RW_WRITER);
1322         kas.a_hat = hat_alloc(&kas);
1323         AS_LOCK_EXIT(&kas);
1323         AS_LOCK_EXIT(&kas, &kas.a_lock);

1325         /*
1326          * Initialize hblk_reserve.
1327          */
1328         ((struct hme_blk *)hblk_reserve)->hblk_nextpa =
1329             va_to_pa((caddr_t)hblk_reserve);

1331 #ifndef UTSB_PHYS
1332         /*
1333          * Reserve some kernel virtual address space for the locked TTEs
1334          * that allow us to probe the TSB from TL>0.
1335          */
1336         utsb_vabase = vmem_xalloc(heap_arena, tsb_slab_size, tsb_slab_size,
1337             0, 0, NULL, NULL, VM_SLEEP);
1338         utsb4m_vabase = vmem_xalloc(heap_arena, tsb_slab_size, tsb_slab_size,
1339             0, 0, NULL, NULL, VM_SLEEP);
1340 #endif

1342 #ifdef VAC
1343         /*
1344          * The big page VAC handling code assumes VAC
1345          * will not be bigger than the smallest big
1346          * page- which is 64K.
1347          */
1348         if (TTEPAGES(TTE64K) < CACHE_NUM_COLOR) {
1349                 cmn_err(CE_PANIC, "VAC too big!");
1350         }
1351 #endif

1353         (void) xhat_init();

1355         uhme_hash_pa = va_to_pa(uhme_hash);
1356         khme_hash_pa = va_to_pa(khme_hash);

1358         /*
1359          * Initialize relocation locks. kpr_suspendlock is held
1360          * at PIL_MAX to prevent interrupts from pinning the holder
1361          * of a suspended TTE which may access it leading to a
1362          * deadlock condition.
1363          */
1364         mutex_init(&kpr_mutex, NULL, MUTEX_DEFAULT, NULL);
1365         mutex_init(&kpr_suspendlock, NULL, MUTEX_SPIN, (void *)PIL_MAX);

1367         /*
1368          * If Shared context support is disabled via /etc/system
1369          * set shctx_on to 0 here if it was set to 1 earlier in boot
1370          * sequence by cpu module initialization code.
1371          */
1372         if (shctx_on && disable_shctx) {
1373                 shctx_on = 0;
1374         }
```



```
1376         if (shctx_on) {
1377                 srd_buckets = kmem_zalloc(SFMMU_MAX_SRD_BUCKETS *
1378                     sizeof (srd_buckets[0]), KM_SLEEP);
1379                 for (i = 0; i < SFMMU_MAX_SRD_BUCKETS; i++) {
1380                         mutex_init(&srd_buckets[i].srdb_lock, NULL,
1381                             MUTEX_DEFAULT, NULL);
1382                 }

1384                 srd_cache = kmem_cache_create("srd_cache", sizeof (sf_srd_t),
1385                     0, sfmmu_srdcache_constructor, sfmmu_srdcache_destructor,
1386                     NULL, NULL, NULL, 0);
1387                 region_cache = kmem_cache_create("region_cache",
1388                     sizeof (sf_region_t), 0, sfmmu_rgncache_constructor,
1389                     sfmmu_rgncache_destructor, NULL, NULL, NULL, 0);
1390                 scd_cache = kmem_cache_create("scd_cache", sizeof (sf_scd_t),
1391                     0, sfmmu_scdcache_constructor,  sfmmu_scdcache_destructor,
1392                     NULL, NULL, NULL, 0);
1393         }

1395         /*
1396          * Pre-allocate hrm_hashtab before enabling the collection of
1397          * refmod statistics.  Allocating on the fly would mean us
1398          * running the risk of suffering recursive mutex enters or
1399          * deadlocks.
1400          */
1401         hrm_hashtab = kmem_zalloc(HRM_HASHSIZE * sizeof (struct hrmstat *),
1402             KM_SLEEP);

1404         /* Allocate per-cpu pending freelist of hmeblks */
1405         cpu_hme_pend = kmem_zalloc((NCPU * sizeof (cpu_hme_pend_t)) + 64,
1406             KM_SLEEP);
1407         cpu_hme_pend = (cpu_hme_pend_t *)P2ROUNDUP(
1408             (uintptr_t)cpu_hme_pend, 64);

1410         for (i = 0; i < NCPU; i++) {
1411                 mutex_init(&cpu_hme_pend[i].chp_mutex, NULL, MUTEX_DEFAULT,
1412                     NULL);
1413         }

1415         if (cpu_hme_pend_thresh == 0) {
1416                 cpu_hme_pend_thresh = CPU_HME_PEND_THRESH;
1417         }
1418 }
______unchanged_portion_omitted_

1451 #define SFMMU_KERNEL_MAXVA \
1452         (kmem64_base ? (uintptr_t)kmem64_end : (SYSLIMIT))

1454 /*
1455  * Allocate a hat structure.
1456  * Called when an address space first uses a hat.
1457  */
1458 struct hat *
1459 hat_alloc(struct as *as)
1460 {
1461         sfmmu_t *sfmmup;
1462         int i;
1463         uint64_t cnum;
1464         extern uint_t get_color_start(struct as *);

1466         ASSERT(AS_WRITE_HELD(as));
1466         ASSERT(AS_WRITE_HELD(as, &as->a_lock));
1467         sfmmup = kmem_cache_alloc(sfmmuid_cache, KM_SLEEP);
1468         sfmmup->sfmmu_as = as;
1469         sfmmup->sfmmu_flags = 0;
```

```
1470         sfmmup->sfmmu_tteflags = 0;
1471         sfmmup->sfmmu_rtteflags = 0;
1472         LOCK_INIT_CLEAR(&sfmmup->sfmmu_ctx_lock);

1474         if (as == &kas) {
1475                 ksfmmup = sfmmup;
1476                 sfmmup->sfmmu_cext = 0;
1477                 cnum = KCONTEXT;

1479                 sfmmup->sfmmu_clrstart = 0;
1480                 sfmmup->sfmmu_tsb = NULL;
1481                 /*
1482                  * hat_kern_setup() will call sfmmu_init_ktsbinfo()
1483                  * to setup tsb_info for ksfmmup.
1484                  */
1485         } else {

1487                 /*
1488                  * Just set to invalid ctx. When it faults, it will
1489                  * get a valid ctx. This would avoid the situation
1490                  * where we get a ctx, but it gets stolen and then
1491                  * we fault when we try to run and so have to get
1492                  * another ctx.
1493                  */
1494                 sfmmup->sfmmu_cext = 0;
1495                 cnum = INVALID_CONTEXT;

1497                 /* initialize original physical page coloring bin */
1498                 sfmmup->sfmmu_clrstart = get_color_start(as);
1499 #ifdef DEBUG
1500                 if (tsb_random_size) {
1501                         uint32_t randval = (uint32_t)gettick() >> 4;
1502                         int size = randval % (tsb_max_growsize + 1);

1504                         /* chose a random tsb size for stress testing */
1505                         (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb, size,
1506                             TSB8K|TSB64K|TSB512K, 0, sfmmup);
1507                 } else
1508 #endif /* DEBUG */
1509                         (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb,
1510                             default_tsb_size,
1511                             TSB8K|TSB64K|TSB512K, 0, sfmmup);
1512                 sfmmup->sfmmu_flags = HAT_SWAPPED | HAT_ALLCTX_INVALID;
1513                 ASSERT(sfmmup->sfmmu_tsb != NULL);
1514         }

1516         ASSERT(max_mmu_ctxdoms > 0);
1517         for (i = 0; i < max_mmu_ctxdoms; i++) {
1518                 sfmmup->sfmmu_ctxs[i].cnum = cnum;
1519                 sfmmup->sfmmu_ctxs[i].gnum = 0;
1520         }

1522         for (i = 0; i < max_mmu_page_sizes; i++) {
1523                 sfmmup->sfmmu_ttecnt[i] = 0;
1524                 sfmmup->sfmmu_scdrttecnt[i] = 0;
1525                 sfmmup->sfmmu_ismttecnt[i] = 0;
1526                 sfmmup->sfmmu_scdismttecnt[i] = 0;
1527                 sfmmup->sfmmu_pgsz[i] = TTE8K;
1528         }
1529         sfmmup->sfmmu_tsb0_4minflcnt = 0;
1530         sfmmup->sfmmu_iblk = NULL;
1531         sfmmup->sfmmu_ismhat = 0;
1532         sfmmup->sfmmu_scdhat = 0;
1533         sfmmup->sfmmu_ismblkpa = (uint64_t)-1;
1534         if (sfmmup == ksfmmup) {
1535                 CPUSET_ALL(sfmmup->sfmmu_cpusran);
```



```
1536         } else {
1537                 CPUSET_ZERO(sfmmup->sfmmu_cpusran);
1538         }
1539         sfmmup->sfmmu_free = 0;
1540         sfmmup->sfmmu_rmstat = 0;
1541         sfmmup->sfmmu_clrbin = sfmmup->sfmmu_clrstart;
1542         sfmmup->sfmmu_xhat_provider = NULL;
1543         cv_init(&sfmmup->sfmmu_tsb_cv, NULL, CV_DEFAULT, NULL);
1544         sfmmup->sfmmu_srdp = NULL;
1545         SF_RGNMAP_ZERO(sfmmup->sfmmu_region_map);
1546         bzero(sfmmup->sfmmu_hmeregion_links, SFMMU_L1_HMERLINKS_SIZE);
1547         sfmmup->sfmmu_scdp = NULL;
1548         sfmmup->sfmmu_scd_link.next = NULL;
1549         sfmmup->sfmmu_scd_link.prev = NULL;
1550         return (sfmmup);
1551 }
______unchanged_portion_omitted_

1913 /*
1914  * Free all the translation resources for the specified address space.
1915  * Called from as_free when an address space is being destroyed.
1916  */
1917 void
1918 hat_free_start(struct hat *sfmmup)
1919 {
1920         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as));
1920         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
1921         ASSERT(sfmmup != ksfmmup);
1922         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

1924         sfmmup->sfmmu_free = 1;
1925         if (sfmmup->sfmmu_scdp != NULL) {
1926                 sfmmu_leave_scd(sfmmup, 0);
1927         }

1929         ASSERT(sfmmup->sfmmu_scdp == NULL);
1930 }
______unchanged_portion_omitted_

2219 /*
2220  * Set up addr to map to page pp with protection prot.
2221  * As an optimization we also load the TSB with the
2222  * corresponding tte but it is no big deal if  the tte gets kicked out.
2223  */
2224 static void
2225 hat_do_memload(struct hat *hat, caddr_t addr, struct page *pp,
2226         uint_t attr, uint_t flags, uint_t rid)
2227 {
2228         tte_t tte;


2231         ASSERT(hat != NULL);
2232         ASSERT(PAGE_LOCKED(pp));
2233         ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));
2234         ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2235         ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2236         SFMMU_VALIDATE_HMERID(hat, rid, addr, MMU_PAGESIZE);

2238         if (PP_ISFREE(pp)) {
2239                 panic("hat_memload: loading a mapping to free page %p",
2240                     (void *)pp);
2241         }

2243         if (hat->sfmmu_xhat_provider) {
2244                 /* no regions for xhats */
2245                 ASSERT(!SFMMU_IS_SHMERID_VALID(rid));
```

```
2246                 XHAT_MEMLOAD(hat, addr, pp, attr, flags);
2247                 return;
2248         }

2250         ASSERT((hat == ksfmmup) || AS_LOCK_HELD(hat->sfmmu_as));
2250         ASSERT((hat == ksfmmup) ||
2251             AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));

2252         if (flags & ~SFMMU_LOAD_ALLFLAG)
2253                 cmn_err(CE_NOTE, "hat_memload: unsupported flags %d",
2254                     flags & ~SFMMU_LOAD_ALLFLAG);

2256         if (hat->sfmmu_rmstat)
2257                 hat_resvstat(MMU_PAGESIZE, hat->sfmmu_as, addr);

2259 #if defined(SF_ERRATA_57)
2260         if ((hat != ksfmmup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2261             (addr < errata57_limit) && (attr & PROT_EXEC) &&
2262             !(flags & HAT_LOAD_SHARE)) {
2263                 cmn_err(CE_WARN, "hat_memload: illegal attempt to make user "
2264                     " page executable");
2265                 attr &= ~PROT_EXEC;
2266         }
2267 #endif

2269         sfmmu_memtte(&tte, pp->p_pagenum, attr, TTE8K);
2270         (void) sfmmu_tteload_array(hat, &tte, addr, &pp, flags, rid);

2272         /*
2273          * Check TSB and TLB page sizes.
2274          */
2275         if ((flags & HAT_LOAD_SHARE) == 0) {
2276                 sfmmu_check_page_sizes(hat, 1);
2277         }
2278 }

2280 /*
2281  * hat_devload can be called to map real memory (e.g.
2282  * /dev/kmem) and even though hat_devload will determine pf is
2283  * for memory, it will be unable to get a shared lock on the
2284  * page (because someone else has it exclusively) and will
2285  * pass dp = NULL.  If tteload doesn't get a non-NULL
2286  * page pointer it can't cache memory.
2287  */
2288 void
2289 hat_devload(struct hat *hat, caddr_t addr, size_t len, pfn_t pfn,
2290         uint_t attr, int flags)
2291 {
2292         tte_t tte;
2293         struct page *pp = NULL;
2294         int use_lgpg = 0;

2296         ASSERT(hat != NULL);

2298         if (hat->sfmmu_xhat_provider) {
2299                 XHAT_DEVLOAD(hat, addr, len, pfn, attr, flags);
2300                 return;
2301         }

2303         ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2304         ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2305         ASSERT((hat == ksfmmup) || AS_LOCK_HELD(hat->sfmmu_as));
2306         ASSERT((hat == ksfmmup) ||
2307             AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));
2306         if (len == 0)
2307                 panic("hat_devload: zero len");
```



```
2308         if (flags & ~SFMMU_LOAD_ALLFLAG)
2309                 cmn_err(CE_NOTE, "hat_devload: unsupported flags %d",
2310                     flags & ~SFMMU_LOAD_ALLFLAG);

2312 #if defined(SF_ERRATA_57)
2313         if ((hat != ksfmmup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2314             (addr < errata57_limit) && (attr & PROT_EXEC) &&
2315             !(flags & HAT_LOAD_SHARE)) {
2316                 cmn_err(CE_WARN, "hat_devload: illegal attempt to make user "
2317                     " page executable");
2318                 attr &= ~PROT_EXEC;
2319         }
2320 #endif

2322         /*
2323          * If it's a memory page find its pp
2324          */
2325         if (!(flags & HAT_LOAD_NOCONSIST) && pf_is_memory(pfn)) {
2326                 pp = page_numtopp_nolock(pfn);
2327                 if (pp == NULL) {
2328                         flags |= HAT_LOAD_NOCONSIST;
2329                 } else {
2330                         if (PP_ISFREE(pp)) {
2331                                 panic("hat_memload: loading "
2332                                     "a mapping to free page %p",
2333                                     (void *)pp);
2334                         }
2335                         if (!PAGE_LOCKED(pp) && !PP_ISNORELOC(pp)) {
2336                                 panic("hat_memload: loading a mapping "
2337                                     "to unlocked relocatable page %p",
2338                                     (void *)pp);
2339                         }
2340                         ASSERT(len == MMU_PAGESIZE);
2341                 }
2342         }

2344         if (hat->sfmmu_rmstat)
2345                 hat_resvstat(len, hat->sfmmu_as, addr);

2347         if (flags & HAT_LOAD_NOCONSIST) {
2348                 attr |= SFMMU_UNCACHEVTTE;
2349                 use_lgpg = 1;
2350         }
2351         if (!pf_is_memory(pfn)) {
2352                 attr |= SFMMU_UNCACHEPTTE | HAT_NOSYNC;
2353                 use_lgpg = 1;
2354                 switch (attr & HAT_ORDER_MASK) {
2355                         case HAT_STRICTORDER:
2356                         case HAT_UNORDERED_OK:
2357                                 /*
2358                                  * we set the side effect bit for all non
2359                                  * memory mappings unless merging is ok
2360                                  */
2361                                 attr |= SFMMU_SIDEFFECT;
2362                                 break;
2363                         case HAT_MERGING_OK:
2364                         case HAT_LOADCACHING_OK:
2365                         case HAT_STORECACHING_OK:
2366                                 break;
2367                         default:
2368                                 panic("hat_devload: bad attr");
2369                                 break;
2370                 }
2371         }
2372         while (len) {
2373                 if (!use_lgpg) {
```

```
2374                        sfmmu_memtte(&tte, pfn, attr, TTE8K);
2375                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2376                            flags, SFMMU_INVALID_SHMERID);
2377                        len -= MMU_PAGESIZE;
2378                        addr += MMU_PAGESIZE;
2379                        pfn++;
2380                        continue;
2381                }
2382                /*
2383                 *  try to use large pages, check va/pa alignments
2384                 *  Note that 32M/256M page sizes are not (yet) supported.
2385                 */
2386                if ((len >= MMU_PAGESIZE4M) &&
2387                    !((uintptr_t)addr & MMU_PAGEOFFSET4M) &&
2388                    !(disable_large_pages & (1 << TTE4M)) &&
2389                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET4M)) {
2390                        sfmmu_memtte(&tte, pfn, attr, TTE4M);
2391                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2392                            flags, SFMMU_INVALID_SHMERID);
2393                        len -= MMU_PAGESIZE4M;
2394                        addr += MMU_PAGESIZE4M;
2395                        pfn += MMU_PAGESIZE4M / MMU_PAGESIZE;
2396                } else if ((len >= MMU_PAGESIZE512K) &&
2397                    !((uintptr_t)addr & MMU_PAGEOFFSET512K) &&
2398                    !(disable_large_pages & (1 << TTE512K)) &&
2399                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET512K)) {
2400                        sfmmu_memtte(&tte, pfn, attr, TTE512K);
2401                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2402                            flags, SFMMU_INVALID_SHMERID);
2403                        len -= MMU_PAGESIZE512K;
2404                        addr += MMU_PAGESIZE512K;
2405                        pfn += MMU_PAGESIZE512K / MMU_PAGESIZE;
2406                } else if ((len >= MMU_PAGESIZE64K) &&
2407                    !((uintptr_t)addr & MMU_PAGEOFFSET64K) &&
2408                    !(disable_large_pages & (1 << TTE64K)) &&
2409                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET64K)) {
2410                        sfmmu_memtte(&tte, pfn, attr, TTE64K);
2411                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2412                            flags, SFMMU_INVALID_SHMERID);
2413                        len -= MMU_PAGESIZE64K;
2414                        addr += MMU_PAGESIZE64K;
2415                        pfn += MMU_PAGESIZE64K / MMU_PAGESIZE;
2416                } else {
2417                        sfmmu_memtte(&tte, pfn, attr, TTE8K);
2418                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2419                            flags, SFMMU_INVALID_SHMERID);
2420                        len -= MMU_PAGESIZE;
2421                        addr += MMU_PAGESIZE;
2422                        pfn++;
2423                }
2424        }

2426        /*
2427         * Check TSB and TLB page sizes.
2428         */
2429        if ((flags & HAT_LOAD_SHARE) == 0) {
2430                sfmmu_check_page_sizes(hat, 1);
2431        }
2432 }
```

unchanged_portion_omitted

```
3959 /*
3960  * Release one hardware address translation lock on the given address range.
3961  */
3962 void
3963 hat_unlock(struct hat *sfmmup, caddr_t addr, size_t len)
```



```
3964 {
3965        struct hmehash_bucket *hmebp;
3966        hmeblk_tag hblktag;
3967        int hmeshift, hashno = 1;
3968        struct hme_blk *hmeblkp, *list = NULL;
3969        caddr_t endaddr;

3971        ASSERT(sfmmup != NULL);
3972        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

3974        ASSERT((sfmmup == ksfmmup) || AS_LOCK_HELD(sfmmup->sfmmu_as));
3976        ASSERT((sfmmup == ksfmmup) ||
3977            AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
3975        ASSERT((len & MMU_PAGEOFFSET) == 0);
3976        endaddr = addr + len;
3977        hblktag.htag_id = sfmmup;
3978        hblktag.htag_rid = SFMMU_INVALID_SHMERID;

3980        /*
3981         * Spitfire supports 4 page sizes.
3982         * Most pages are expected to be of the smallest page size (8K) and
3983         * these will not need to be rehashed. 64K pages also don't need to be
3984         * rehashed because an hmeblk spans 64K of address space. 512K pages
3985         * might need 1 rehash and and 4M pages might need 2 rehashes.
3986         */
3987        while (addr < endaddr) {
3988                hmeshift = HME_HASH_SHIFT(hashno);
3989                hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
3990                hblktag.htag_rehash = hashno;
3991                hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

3993                SFMMU_HASH_LOCK(hmebp);

3995                HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
3996                if (hmeblkp != NULL) {
3997                        ASSERT(!hmeblkp->hblk_shared);
3998                        /*
3999                         * If we encounter a shadow hmeblk then
4000                         * we know there are no valid hmeblks mapping
4001                         * this address at this size or larger.
4002                         * Just increment address by the smallest
4003                         * page size.
4004                         */
4005                        if (hmeblkp->hblk_shw_bit) {
4006                                addr += MMU_PAGESIZE;
4007                        } else {
4008                                addr = sfmmu_hblk_unlock(hmeblkp, addr,
4009                                    endaddr);
4010                        }
4011                        SFMMU_HASH_UNLOCK(hmebp);
4012                        hashno = 1;
4013                        continue;
4014                }
4015                SFMMU_HASH_UNLOCK(hmebp);

4017                if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
4018                        /*
4019                         * We have traversed the whole list and rehashed
4020                         * if necessary without finding the address to unlock
4021                         * which should never happen.
4022                         */
4023                        panic("sfmmu_unlock: addr not found. "
4024                            "addr %p hat %p", (void *)addr, (void *)sfmmup);
4025                } else {
4026                        hashno++;
4027                }
```

```
4028         }

4030         sfmmu_hblks_list_purge(&list, 0);
4031 }
_____unchanged_portion_omitted_

4751 /*
4752  * hat_probe returns 1 if the translation for the address 'addr' is
4753  * loaded, zero otherwise.
4754  *
4755  * hat_probe should be used only for advisorary purposes because it may
4756  * occasionally return the wrong value. The implementation must guarantee that
4757  * returning the wrong value is a very rare event. hat_probe is used
4758  * to implement optimizations in the segment drivers.
4759  *
4760  */
4761 int
4762 hat_probe(struct hat *sfmmup, caddr_t addr)
4763 {
4764         pfn_t pfn;
4765         tte_t tte;

4767         ASSERT(sfmmup != NULL);
4768         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4770         ASSERT((sfmmup == ksfmmup) || AS_LOCK_HELD(sfmmup->sfmmu_as));
4773         ASSERT((sfmmup == ksfmmup) ||
4774             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

4772         if (sfmmup == ksfmmup) {
4773                 while ((pfn = sfmmu_vatopfn(addr, sfmmup, &tte))
4774                     == PFN_SUSPENDED) {
4775                         sfmmu_vatopfn_suspended(addr, sfmmup, &tte);
4776                 }
4777         } else {
4778                 pfn = sfmmu_uvatopfn(addr, sfmmup, NULL);
4779         }

4781         if (pfn != PFN_INVALID)
4782                 return (1);
4783         else
4784                 return (0);
4785 }
_____unchanged_portion_omitted_

4902 /*
4903  * Change attributes on an address range to that specified by attr and mode.
4904  */
4905 static void
4906 sfmmu_chgattr(struct hat *sfmmup, caddr_t addr, size_t len, uint_t attr,
4907         int mode)
4908 {
4909         struct hmehash_bucket *hmebp;
4910         hmeblk_tag hblktag;
4911         int hmeshift, hashno = 1;
4912         struct hme_blk *hmeblkp, *list = NULL;
4913         caddr_t endaddr;
4914         cpuset_t cpuset;
4915         demap_range_t dmr;

4917         CPUSET_ZERO(cpuset);

4919         ASSERT((sfmmup == ksfmmup) || AS_LOCK_HELD(sfmmup->sfmmu_as));
4923         ASSERT((sfmmup == ksfmmup) ||
4924             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
4920         ASSERT((len & MMU_PAGEOFFSET) == 0);
```



```
4921         ASSERT(((uintptr_t)addr & MMU_PAGEOFFSET) == 0);

4923         if ((attr & PROT_USER) && (mode != SFMMU_CLRATTR) &&
4924             ((addr + len) > (caddr_t)USERLIMIT)) {
4925                 panic("user addr %p in kernel space",
4926                     (void *)addr);
4927         }

4929         endaddr = addr + len;
4930         hblktag.htag_id = sfmmup;
4931         hblktag.htag_rid = SFMMU_INVALID_SHMERID;
4932         DEMAP_RANGE_INIT(sfmmup, &dmr);

4934         while (addr < endaddr) {
4935                 hmeshift = HME_HASH_SHIFT(hashno);
4936                 hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
4937                 hblktag.htag_rehash = hashno;
4938                 hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

4940                 SFMMU_HASH_LOCK(hmebp);

4942                 HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
4943                 if (hmeblkp != NULL) {
4944                         ASSERT(!hmeblkp->hblk_shared);
4945                         /*
4946                          * We've encountered a shadow hmeblk so skip the range
4947                          * of the next smaller mapping size.
4948                          */
4949                         if (hmeblkp->hblk_shw_bit) {
4950                                 ASSERT(sfmmup != ksfmmup);
4951                                 ASSERT(hashno > 1);
4952                                 addr = (caddr_t)P2END((uintptr_t)addr,
4953                                     TTEBYTES(hashno - 1));
4954                         } else {
4955                                 addr = sfmmu_hblk_chgattr(sfmmup,
4956                                     hmeblkp, addr, endaddr, &dmr, attr, mode);
4957                         }
4958                         SFMMU_HASH_UNLOCK(hmebp);
4959                         hashno = 1;
4960                         continue;
4961                 }
4962                 SFMMU_HASH_UNLOCK(hmebp);

4964                 if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
4965                         /*
4966                          * We have traversed the whole list and rehashed
4967                          * if necessary without finding the address to chgattr.
4968                          * This is ok, so we increment the address by the
4969                          * smallest hmeblk range for kernel mappings or for
4970                          * user mappings with no large pages, and the largest
4971                          * hmeblk range, to account for shadow hmeblks, for
4972                          * user mappings with large pages and continue.
4973                          */
4974                         if (sfmmup == ksfmmup)
4975                                 addr = (caddr_t)P2END((uintptr_t)addr,
4976                                     TTEBYTES(1));
4977                         else
4978                                 addr = (caddr_t)P2END((uintptr_t)addr,
4979                                     TTEBYTES(hashno));
4980                         hashno = 1;
4981                 } else {
4982                         hashno++;
4983                 }
4984         }

4986         sfmmu_hblks_list_purge(&list, 0);
```

```
4987         DEMAP_RANGE_FLUSH(&dmr);
4988         cpuset = sfmmup->sfmmu_cpusran;
4989         xt_sync(cpuset);
4990 }
_____unchanged_portion_omitted_

5665 /*
5666  * Unload all the mappings in the range [addr..addr+len). addr and len must
5667  * be MMU_PAGESIZE aligned.
5668  */

5670 extern struct seg *segkmap;
5671 #define ISSEGKMAP(sfmmup, addr) (sfmmup == ksfmmup && \
5672 segkmap->s_base <= (addr) && (addr) < (segkmap->s_base + segkmap->s_size))


5675 void
5676 hat_unload_callback(
5677         struct hat *sfmmup,
5678         caddr_t addr,
5679         size_t len,
5680         uint_t flags,
5681         hat_callback_t *callback)
5682 {
5683         struct hmehash_bucket *hmebp;
5684         hmeblk_tag hblktag;
5685         int hmeshift, hashno, iskernel;
5686         struct hme_blk *hmeblkp, *pr_hblk, *list = NULL;
5687         caddr_t endaddr;
5688         cpuset_t cpuset;
5689         int addr_count = 0;
5690         int a;
5691         caddr_t cb_start_addr[MAX_CB_ADDR];
5692         caddr_t cb_end_addr[MAX_CB_ADDR];
5693         int issegkmap = ISSEGKMAP(sfmmup, addr);
5694         demap_range_t dmr, *dmrp;

5696         if (sfmmup->sfmmu_xhat_provider) {
5697                 XHAT_UNLOAD_CALLBACK(sfmmup, addr, len, flags, callback);
5698                 return;
5699         } else {
5700                 /*
5701                  * This must be a CPU HAT. If the address space has
5702                  * XHATs attached, unload the mappings for all of them,
5703                  * just in case
5704                  */
5705                 ASSERT(sfmmup->sfmmu_as != NULL);
5706                 if (sfmmup->sfmmu_as->a_xhat != NULL)
5707                         xhat_unload_callback_all(sfmmup->sfmmu_as, addr,
5708                             len, flags, callback);
5709         }

5711         ASSERT((sfmmup == ksfmmup) || (flags & HAT_UNLOAD_OTHER) || \
5712             AS_LOCK_HELD(sfmmup->sfmmu_as));
5717             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

5714         ASSERT(sfmmup != NULL);
5715         ASSERT((len & MMU_PAGEOFFSET) == 0);
5716         ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));

5718         /*
5719          * Probing through a large VA range (say 63 bits) will be slow, even
5720          * at 4 Meg steps between the probes. So, when the virtual address range
5721          * is very large, search the HME entries for what to unload.
5722          *
5723          *      len >> TTE_PAGE_SHIFT(TTE4M) is the # of 4Meg probes we'd need
```



```
5724          *
5725          *      UHMEHASH_SZ is number of hash buckets to examine
5726          *
5727          */
5728         if (sfmmup != KHATID && (len >> TTE_PAGE_SHIFT(TTE4M)) > UHMEHASH_SZ) {
5729                 hat_unload_large_virtual(sfmmup, addr, len, flags, callback);
5730                 return;
5731         }

5733         CPUSET_ZERO(cpuset);

5735         /*
5736          * If the process is exiting, we can save a lot of fuss since
5737          * we'll flush the TLB when we free the ctx anyway.
5738          */
5739         if (sfmmup->sfmmu_free) {
5740                 dmrp = NULL;
5741         } else {
5742                 dmrp = &dmr;
5743                 DEMAP_RANGE_INIT(sfmmup, dmrp);
5744         }

5746         endaddr = addr + len;
5747         hblktag.htag_id = sfmmup;
5748         hblktag.htag_rid = SFMMU_INVALID_SHMERID;

5750         /*
5751          * It is likely for the vm to call unload over a wide range of
5752          * addresses that are actually very sparsely populated by
5753          * translations.  In order to speed this up the sfmmu hat supports
5754          * the concept of shadow hmeblks. Dummy large page hmeblks that
5755          * correspond to actual small translations are allocated at tteload
5756          * time and are referred to as shadow hmeblks.  Now, during unload
5757          * time, we first check if we have a shadow hmeblk for that
5758          * translation.  The absence of one means the corresponding address
5759          * range is empty and can be skipped.
5760          *
5761          * The kernel is an exception to above statement and that is why
5762          * we don't use shadow hmeblks and hash starting from the smallest
5763          * page size.
5764          */
5765         if (sfmmup == KHATID) {
5766                 iskernel = 1;
5767                 hashno = TTE64K;
5768         } else {
5769                 iskernel = 0;
5770                 if (mmu_page_sizes == max_mmu_page_sizes) {
5771                         hashno = TTE256M;
5772                 } else {
5773                         hashno = TTE4M;
5774                 }
5775         }
5776         while (addr < endaddr) {
5777                 hmeshift = HME_HASH_SHIFT(hashno);
5778                 hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
5779                 hblktag.htag_rehash = hashno;
5780                 hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

5782                 SFMMU_HASH_LOCK(hmebp);

5784                 HME_HASH_SEARCH_PREV(hmebp, hblktag, hmeblkp, pr_hblk, &list);
5785                 if (hmeblkp == NULL) {
5786                         /*
5787                          * didn't find an hmeblk. skip the appropiate
5788                          * address range.
5789                          */
```

```
5790                        SFMMU_HASH_UNLOCK(hmebp);
5791                        if (iskernel) {
5792                                if (hashno < mmu_hashcnt) {
5793                                        hashno++;
5794                                        continue;
5795                                } else {
5796                                        hashno = TTE64K;
5797                                        addr = (caddr_t)roundup((uintptr_t)addr
5798                                            + 1, MMU_PAGESIZE64K);
5799                                        continue;
5800                                }
5801                        }
5802                        addr = (caddr_t)roundup((uintptr_t)addr + 1,
5803                            (1 << hmeshift));
5804                        if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5805                                ASSERT(hashno == TTE64K);
5806                                continue;
5807                        }
5808                        if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5809                                hashno = TTE512K;
5810                                continue;
5811                        }
5812                        if (mmu_page_sizes == max_mmu_page_sizes) {
5813                                if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5814                                        hashno = TTE4M;
5815                                        continue;
5816                                }
5817                                if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5818                                        hashno = TTE32M;
5819                                        continue;
5820                                }
5821                                hashno = TTE256M;
5822                                continue;
5823                        } else {
5824                                hashno = TTE4M;
5825                                continue;
5826                        }
5827                }
5828                ASSERT(hmeblkp);
5829                ASSERT(!hmeblkp->hblk_shared);
5830                if (!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5831                        /*
5832                         * If the valid count is zero we can skip the range
5833                         * mapped by this hmeblk.
5834                         * We free hblks in the case of HAT_UNMAP.  HAT_UNMAP
5835                         * is used by segment drivers as a hint
5836                         * that the mapping resource won't be used any longer.
5837                         * The best example of this is during exit().
5838                         */
5839                        addr = (caddr_t)roundup((uintptr_t)addr + 1,
5840                            get_hblk_span(hmeblkp));
5841                        if ((flags & HAT_UNLOAD_UNMAP) ||
5842                            (iskernel && !issegkmap)) {
5843                                sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
5844                                    &list, 0);
5845                        }
5846                        SFMMU_HASH_UNLOCK(hmebp);

5848                        if (iskernel) {
5849                                hashno = TTE64K;
5850                                continue;
5851                        }
5852                        if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5853                                ASSERT(hashno == TTE64K);
5854                                continue;
5855                        }
```



```
5856                        if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5857                                hashno = TTE512K;
5858                                continue;
5859                        }
5860                        if (mmu_page_sizes == max_mmu_page_sizes) {
5861                                if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5862                                        hashno = TTE4M;
5863                                        continue;
5864                                }
5865                                if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5866                                        hashno = TTE32M;
5867                                        continue;
5868                                }
5869                                hashno = TTE256M;
5870                                continue;
5871                        } else {
5872                                hashno = TTE4M;
5873                                continue;
5874                        }
5875                }
5876                if (hmeblkp->hblk_shw_bit) {
5877                        /*
5878                         * If we encounter a shadow hmeblk we know there is
5879                         * smaller sized hmeblks mapping the same address space.
5880                         * Decrement the hash size and rehash.
5881                         */
5882                        ASSERT(sfmmup != KHATID);
5883                        hashno--;
5884                        SFMMU_HASH_UNLOCK(hmebp);
5885                        continue;
5886                }

5888                /*
5889                 * track callback address ranges.
5890                 * only start a new range when it's not contiguous
5891                 */
5892                if (callback != NULL) {
5893                        if (addr_count > 0 &&
5894                            addr == cb_end_addr[addr_count - 1])
5895                                --addr_count;
5896                        else
5897                                cb_start_addr[addr_count] = addr;
5898                }

5900                addr = sfmmu_hblk_unload(sfmmup, hmeblkp, addr, endaddr,
5901                    dmrp, flags);

5903                if (callback != NULL)
5904                        cb_end_addr[addr_count++] = addr;

5906                if (((flags & HAT_UNLOAD_UNMAP) || (iskernel && !issegkmap)) &&
5907                    !hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5908                        sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk, &list, 0);
5909                }
5910                SFMMU_HASH_UNLOCK(hmebp);

5912                /*
5913                 * Notify our caller as to exactly which pages
5914                 * have been unloaded. We do these in clumps,
5915                 * to minimize the number of xt_sync()s that need to occur.
5916                 */
5917                if (callback != NULL && addr_count == MAX_CB_ADDR) {
5918                        if (dmrp != NULL) {
5919                                DEMAP_RANGE_FLUSH(dmrp);
5920                                cpuset = sfmmup->sfmmu_cpusran;
5921                                xt_sync(cpuset);
```

```
5922                        }

5924                        for (a = 0; a < MAX_CB_ADDR; ++a) {
5925                                callback->hcb_start_addr = cb_start_addr[a];
5926                                callback->hcb_end_addr = cb_end_addr[a];
5927                                callback->hcb_function(callback);
5928                        }
5929                        addr_count = 0;
5930                }
5931                if (iskernel) {
5932                        hashno = TTE64K;
5933                        continue;
5934                }
5935                if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5936                        ASSERT(hashno == TTE64K);
5937                        continue;
5938                }
5939                if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5940                        hashno = TTE512K;
5941                        continue;
5942                }
5943                if (mmu_page_sizes == max_mmu_page_sizes) {
5944                        if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5945                                hashno = TTE4M;
5946                                continue;
5947                        }
5948                        if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5949                                hashno = TTE32M;
5950                                continue;
5951                        }
5952                        hashno = TTE256M;
5953                } else {
5954                        hashno = TTE4M;
5955                }
5956        }

5958        sfmmu_hblks_list_purge(&list, 0);
5959        if (dmrp != NULL) {
5960                DEMAP_RANGE_FLUSH(dmrp);
5961                cpuset = sfmmup->sfmmu_cpusran;
5962                xt_sync(cpuset);
5963        }
5964        if (callback && addr_count != 0) {
5965                for (a = 0; a < addr_count; ++a) {
5966                        callback->hcb_start_addr = cb_start_addr[a];
5967                        callback->hcb_end_addr = cb_end_addr[a];
5968                        callback->hcb_function(callback);
5969                }
5970        }

5972        /*
5973         * Check TSB and TLB page sizes if the process isn't exiting.
5974         */
5975        if (!sfmmup->sfmmu_free)
5976                sfmmu_check_page_sizes(sfmmup, 0);
5977 }
```

_____unchanged_portion_omitted_

```
6313 /*
6314  * Synchronize all the mappings in the range [addr..addr+len).
6315  * Can be called with clearflag having two states:
6316  * HAT_SYNC_DONTZERO means just return the rm stats
6317  * HAT_SYNC_ZERORM means zero rm bits in the tte and return the stats
6318  */
6319 void
6320 hat_sync(struct hat *sfmmup, caddr_t addr, size_t len, uint_t clearflag)
```



```
6321 {
6322        struct hmehash_bucket *hmebp;
6323        hmeblk_tag hblktag;
6324        int hmeshift, hashno = 1;
6325        struct hme_blk *hmeblkp, *list = NULL;
6326        caddr_t endaddr;
6327        cpuset_t cpuset;

6329        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
6330        ASSERT((sfmmup == ksfmmup) || AS_LOCK_HELD(sfmmup->sfmmu_as));
6335        ASSERT((sfmmup == ksfmmup) ||
6336            AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
6331        ASSERT((len & MMU_PAGEOFFSET) == 0);
6332        ASSERT((clearflag == HAT_SYNC_DONTZERO) ||
6333            (clearflag == HAT_SYNC_ZERORM));

6335        CPUSET_ZERO(cpuset);

6337        endaddr = addr + len;
6338        hblktag.htag_id = sfmmup;
6339        hblktag.htag_rid = SFMMU_INVALID_SHMERID;

6341        /*
6342         * Spitfire supports 4 page sizes.
6343         * Most pages are expected to be of the smallest page
6344         * size (8K) and these will not need to be rehashed. 64K
6345         * pages also don't need to be rehashed because the an hmeblk
6346         * spans 64K of address space. 512K pages might need 1 rehash and
6347         * and 4M pages 2 rehashes.
6348         */
6349        while (addr < endaddr) {
6350                hmeshift = HME_HASH_SHIFT(hashno);
6351                hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
6352                hblktag.htag_rehash = hashno;
6353                hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

6355                SFMMU_HASH_LOCK(hmebp);

6357                HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
6358                if (hmeblkp != NULL) {
6359                        ASSERT(!hmeblkp->hblk_shared);
6360                        /*
6361                         * We've encountered a shadow hmeblk so skip the range
6362                         * of the next smaller mapping size.
6363                         */
6364                        if (hmeblkp->hblk_shw_bit) {
6365                                ASSERT(sfmmup != ksfmmup);
6366                                ASSERT(hashno > 1);
6367                                addr = (caddr_t)P2END((uintptr_t)addr,
6368                                    TTEBYTES(hashno - 1));
6369                        } else {
6370                                addr = sfmmu_hblk_sync(sfmmup, hmeblkp,
6371                                    addr, endaddr, clearflag);
6372                        }
6373                        SFMMU_HASH_UNLOCK(hmebp);
6374                        hashno = 1;
6375                        continue;
6376                }
6377                SFMMU_HASH_UNLOCK(hmebp);

6379                if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
6380                        /*
6381                         * We have traversed the whole list and rehashed
6382                         * if necessary without finding the address to sync.
6383                         * This is ok so we increment the address by the
6384                         * smallest hmeblk range for kernel mappings and the
```

```
6385                         * largest hmeblk range, to account for shadow hmeblks,
6386                         * for user mappings and continue.
6387                         */
6388                        if (sfmmup == ksfmmup)
6389                                addr = (caddr_t)P2END((uintptr_t)addr,
6390                                    TTEBYTES(1));
6391                        else
6392                                addr = (caddr_t)P2END((uintptr_t)addr,
6393                                    TTEBYTES(hashno));
6394                        hashno = 1;
6395                } else {
6396                        hashno++;
6397                }
6398        }
6399        sfmmu_hblks_list_purge(&list, 0);
6400        cpuset = sfmmup->sfmmu_cpusran;
6401        xt_sync(cpuset);
6402 }
_____unchanged_portion_omitted_
7959 #endif  /* DEBUG */

7961 /*
7962  * Returns a page frame number for a given virtual address.
7963  * Returns PFN_INVALID to indicate an invalid mapping
7964  */
7965 pfn_t
7966 hat_getpfnum(struct hat *hat, caddr_t addr)
7967 {
7968        pfn_t pfn;
7969        tte_t tte;

7971        /*
7972         * We would like to
7973         * ASSERT(AS_LOCK_HELD(as));
7979         * ASSERT(AS_LOCK_HELD(as, &as->a_lock));
7974         * but we can't because the iommu driver will call this
7975         * routine at interrupt time and it can't grab the as lock
7976         * or it will deadlock: A thread could have the as lock
7977         * and be waiting for io.  The io can't complete
7978         * because the interrupt thread is blocked trying to grab
7979         * the as lock.
7980         */

7982        ASSERT(hat->sfmmu_xhat_provider == NULL);

7984        if (hat == ksfmmup) {
7985                if (IS_KMEM_VA_LARGEPAGE(addr)) {
7986                        ASSERT(segkmem_lpszc > 0);
7987                        pfn = sfmmu_kvaszc2pfn(addr, segkmem_lpszc);
7988                        if (pfn != PFN_INVALID) {
7989                                sfmmu_check_kpfn(pfn);
7990                                return (pfn);
7991                        }
7992                } else if (segkpm && IS_KPM_ADDR(addr)) {
7993                        return (sfmmu_kpm_vatopfn(addr));
7994                }
7995                while ((pfn = sfmmu_vatopfn(addr, ksfmmup, &tte))
7996                    == PFN_SUSPENDED) {
7997                        sfmmu_vatopfn_suspended(addr, ksfmmup, &tte);
7998                }
7999                sfmmu_check_kpfn(pfn);
8000                return (pfn);
8001        } else {
8002                return (sfmmu_uvatopfn(addr, hat, NULL));
8003        }
8004 }
_____unchanged_portion_omitted_
```



```
13975 /*
13976  * The caller makes sure hat_join_region()/hat_leave_region() can't be called
13977  * at the same time for the same process and address range. This is ensured by
13978  * the fact that address space is locked as writer when a process joins the
13979  * regions. Therefore there's no need to hold an srd lock during the entire
13980  * execution of hat_join_region()/hat_leave_region().
13981  */

13983 #define RGN_HASH_FUNCTION(obj)  (((((uintptr_t)(obj)) >> 4) ^ \
13984                                     (((uintptr_t)(obj)) >> 11)) & \
13985                                         srd_rgn_hashmask)
13986 /*
13987  * This routine implements the shared context functionality required when
13988  * attaching a segment to an address space. It must be called from
13989  * hat_share() for D(ISM) segments and from segvn_create() for segments
13990  * with the MAP_PRIVATE and MAP_TEXT flags set. It returns a region_cookie
13991  * which is saved in the private segment data for hme segments and
13992  * the ism_map structure for ism segments.
13993  */
13994 hat_region_cookie_t
13995 hat_join_region(struct hat *sfmmup,
13996         caddr_t r_saddr,
13997         size_t r_size,
13998         void *r_obj,
13999         u_offset_t r_objoff,
14000         uchar_t r_perm,
14001         uchar_t r_pgszc,
14002         hat_rgn_cb_func_t r_cb_function,
14003         uint_t flags)
14004 {
14005         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14006         uint_t rhash;
14007         uint_t rid;
14008         hatlock_t *hatlockp;
14009         sf_region_t *rgnp;
14010         sf_region_t *new_rgnp = NULL;
14011         int i;
14012         uint16_t *nextidp;
14013         sf_region_t **freelistp;
14014         int maxids;
14015         sf_region_t **rarrp;
14016         uint16_t *busyrgnsp;
14017         ulong_t rttecnt;
14018         uchar_t tteflag;
14019         uchar_t r_type = flags & HAT_REGION_TYPE_MASK;
14020         int text = (r_type == HAT_REGION_TEXT);

14022         if (srdp == NULL || r_size == 0) {
14023                 return (HAT_INVALID_REGION_COOKIE);
14024         }

14026         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
14027         ASSERT(sfmmup != ksfmmup);
14028         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as));
14034         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
14029         ASSERT(srdp->srd_refcnt > 0);
14030         ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
14031         ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
14032         ASSERT(r_pgszc < mmu_page_sizes);
14033         if (!IS_P2ALIGNED(r_saddr, TTEBYTES(r_pgszc)) ||
14034             !IS_P2ALIGNED(r_size, TTEBYTES(r_pgszc))) {
14035                 panic("hat_join_region: region addr or size is not aligned\n");
14036         }
```

```
14039         r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
14040             SFMMU_REGION_HME;
14041         /*
14042          * Currently only support shared hmes for the read only main text
14043          * region.
14044          */
14045         if (r_type == SFMMU_REGION_HME && ((r_obj != srdp->srd_evp) ||
14046             (r_perm & PROT_WRITE))) {
14047                 return (HAT_INVALID_REGION_COOKIE);
14048         }

14050         rhash = RGN_HASH_FUNCTION(r_obj);

14052         if (r_type == SFMMU_REGION_ISM) {
14053                 nextidp = &srdp->srd_next_ismrid;
14054                 freelistp = &srdp->srd_ismrgnfree;
14055                 maxids = SFMMU_MAX_ISM_REGIONS;
14056                 rarrp = srdp->srd_ismrgnp;
14057                 busyrgnsp = &srdp->srd_ismbusyrgns;
14058         } else {
14059                 nextidp = &srdp->srd_next_hmerid;
14060                 freelistp = &srdp->srd_hmergnfree;
14061                 maxids = SFMMU_MAX_HME_REGIONS;
14062                 rarrp = srdp->srd_hmergnp;
14063                 busyrgnsp = &srdp->srd_hmebusyrgns;
14064         }

14066         mutex_enter(&srdp->srd_mutex);

14068         for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
14069             rgnp = rgnp->rgn_hash) {
14070                 if (rgnp->rgn_saddr == r_saddr && rgnp->rgn_size == r_size &&
14071                     rgnp->rgn_obj == r_obj && rgnp->rgn_objoff == r_objoff &&
14072                     rgnp->rgn_perm == r_perm && rgnp->rgn_pgszc == r_pgszc) {
14073                         break;
14074                 }
14075         }

14077 rfound:
14078         if (rgnp != NULL) {
14079                 ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14080                 ASSERT(rgnp->rgn_cb_function == r_cb_function);
14081                 ASSERT(rgnp->rgn_refcnt >= 0);
14082                 rid = rgnp->rgn_id;
14083                 ASSERT(rid < maxids);
14084                 ASSERT(rarrp[rid] == rgnp);
14085                 ASSERT(rid < *nextidp);
14086                 atomic_inc_32((volatile uint_t *)&rgnp->rgn_refcnt);
14087                 mutex_exit(&srdp->srd_mutex);
14088                 if (new_rgnp != NULL) {
14089                         kmem_cache_free(region_cache, new_rgnp);
14090                 }
14091                 if (r_type == SFMMU_REGION_HME) {
14092                         int myjoin =
14093                             (sfmmup == astosfmmu(curthread->t_procp->p_as));

14095                         sfmmu_link_to_hmeregion(sfmmup, rgnp);
14096                         /*
14097                          * bitmap should be updated after linking sfmmu on
14098                          * region list so that pageunload() doesn't skip
14099                          * TSB/TLB flush. As soon as bitmap is updated another
14100                          * thread in this process can already start accessing
14101                          * this region.
14102                          */
14103                         /*
14104                          * Normally ttecnt accounting is done as part of
```



```
14105                          * pagefault handling. But a process may not take any
14106                          * pagefaults on shared hmeblks created by some other
14107                          * process. To compensate for this assume that the
14108                          * entire region will end up faulted in using
14109                          * the region's pagesize.
14110                          *
14111                          */
14112                         if (r_pgszc > TTE8K) {
14113                                 tteflag = 1 << r_pgszc;
14114                                 if (disable_large_pages & tteflag) {
14115                                         tteflag = 0;
14116                                 }
14117                         } else {
14118                                 tteflag = 0;
14119                         }
14120                         if (tteflag && !(sfmmup->sfmmu_rtteflags & tteflag)) {
14121                                 hatlockp = sfmmu_hat_enter(sfmmup);
14122                                 sfmmup->sfmmu_rtteflags |= tteflag;
14123                                 sfmmu_hat_exit(hatlockp);
14124                         }
14125                         hatlockp = sfmmu_hat_enter(sfmmup);

14127                         /*
14128                          * Preallocate 1/4 of ttecnt's in 8K TSB for >= 4M
14129                          * region to allow for large page allocation failure.
14130                          */
14131                         if (r_pgszc >= TTE4M) {
14132                                 sfmmup->sfmmu_tsb0_4minflcnt +=
14133                                     r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14134                         }

14136                         /* update sfmmu_ttecnt with the shme rgn ttecnt */
14137                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14138                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
14139                             rttecnt);

14141                         if (text && r_pgszc >= TTE4M &&
14142                             (tteflag || ((disable_large_pages >> TTE4M) &
14143                             ((1 << (r_pgszc - TTE4M + 1)) - 1))) &&
14144                             !SFMMU_FLAGS_ISSET(sfmmup, HAT_4MTEXT_FLAG)) {
14145                                 SFMMU_FLAGS_SET(sfmmup, HAT_4MTEXT_FLAG);
14146                         }

14148                         sfmmu_hat_exit(hatlockp);
14149                         /*
14150                          * On Panther we need to make sure TLB is programmed
14151                          * to accept 32M/256M pages.  Call
14152                          * sfmmu_check_page_sizes() now to make sure TLB is
14153                          * setup before making hmeregions visible to other
14154                          * threads.
14155                          */
14156                         sfmmu_check_page_sizes(sfmmup, 1);
14157                         hatlockp = sfmmu_hat_enter(sfmmup);
14158                         SF_RGNMAP_ADD(sfmmup->sfmmu_hmeregion_map, rid);

14160                         /*
14161                          * if context is invalid tsb miss exception code will
14162                          * call sfmmu_check_page_sizes() and update tsbmiss
14163                          * area later.
14164                          */
14165                         kpreempt_disable();
14166                         if (myjoin &&
14167                             (sfmmup->sfmmu_ctxs[CPU_MMU_IDX(CPU)].cnum
14168                             != INVALID_CONTEXT)) {
14169                                 struct tsbmiss *tsbmp;
```

```
14171                                   tsbmp = &tsbmiss_area[CPU->cpu_id];
14172                                   ASSERT(sfmmup == tsbmp->usfmmup);
14173                                   BT_SET(tsbmp->shmermap, rid);
14174                                   if (r_pgszc > TTE64K) {
14175                                           tsbmp->uhat_rtteflags |= tteflag;
14176                                   }

14178                           }
14179                           kpreempt_enable();

14181                           sfmmu_hat_exit(hatlockp);
14182                           ASSERT((hat_region_cookie_t)((uint64_t)rid) !=
14183                               HAT_INVALID_REGION_COOKIE);
14184                   } else {
14185                           hatlockp = sfmmu_hat_enter(sfmmup);
14186                           SF_RGNMAP_ADD(sfmmup->sfmmu_ismregion_map, rid);
14187                           sfmmu_hat_exit(hatlockp);
14188                   }
14189                   ASSERT(rid < maxids);

14191                   if (r_type == SFMMU_REGION_ISM) {
14192                           sfmmu_find_scd(sfmmup);
14193                   }
14194                   return ((hat_region_cookie_t)((uint64_t)rid));
14195           }

14197           ASSERT(new_rgnp == NULL);

14199           if (*busyrgnsp >= maxids) {
14200                   mutex_exit(&srdp->srd_mutex);
14201                   return (HAT_INVALID_REGION_COOKIE);
14202           }

14204           ASSERT(MUTEX_HELD(&srdp->srd_mutex));
14205           if (*freelistp != NULL) {
14206                   rgnp = *freelistp;
14207                   *freelistp = rgnp->rgn_next;
14208                   ASSERT(rgnp->rgn_id < *nextidp);
14209                   ASSERT(rgnp->rgn_id < maxids);
14210                   ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
14211                   ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK)
14212                       == r_type);
14213                   ASSERT(rarrp[rgnp->rgn_id] == rgnp);
14214                   ASSERT(rgnp->rgn_hmeflags == 0);
14215           } else {
14216                   /*
14217                    * release local locks before memory allocation.
14218                    */
14219                   mutex_exit(&srdp->srd_mutex);

14221                   new_rgnp = kmem_cache_alloc(region_cache, KM_SLEEP);

14223                   mutex_enter(&srdp->srd_mutex);
14224                   for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
14225                       rgnp = rgnp->rgn_hash) {
14226                           if (rgnp->rgn_saddr == r_saddr &&
14227                               rgnp->rgn_size == r_size &&
14228                               rgnp->rgn_obj == r_obj &&
14229                               rgnp->rgn_objoff == r_objoff &&
14230                               rgnp->rgn_perm == r_perm &&
14231                               rgnp->rgn_pgszc == r_pgszc) {
14232                                   break;
14233                           }
14234                   }
14235                   if (rgnp != NULL) {
14236                           goto rfound;
```



```
14237                   }

14239                   if (*nextidp >= maxids) {
14240                           mutex_exit(&srdp->srd_mutex);
14241                           goto fail;
14242                   }
14243                   rgnp = new_rgnp;
14244                   new_rgnp = NULL;
14245                   rgnp->rgn_id = (*nextidp)++;
14246                   ASSERT(rgnp->rgn_id < maxids);
14247                   ASSERT(rarrp[rgnp->rgn_id] == NULL);
14248                   rarrp[rgnp->rgn_id] = rgnp;
14249           }

14251           ASSERT(rgnp->rgn_sfmmu_head == NULL);
14252           ASSERT(rgnp->rgn_hmeflags == 0);
14253 #ifdef DEBUG
14254           for (i = 0; i < MMU_PAGE_SIZES; i++) {
14255                   ASSERT(rgnp->rgn_ttecnt[i] == 0);
14256           }
14257 #endif
14258           rgnp->rgn_saddr = r_saddr;
14259           rgnp->rgn_size = r_size;
14260           rgnp->rgn_obj = r_obj;
14261           rgnp->rgn_objoff = r_objoff;
14262           rgnp->rgn_perm = r_perm;
14263           rgnp->rgn_pgszc = r_pgszc;
14264           rgnp->rgn_flags = r_type;
14265           rgnp->rgn_refcnt = 0;
14266           rgnp->rgn_cb_function = r_cb_function;
14267           rgnp->rgn_hash = srdp->srd_rgnhash[rhash];
14268           srdp->srd_rgnhash[rhash] = rgnp;
14269           (*busyrgnsp)++;
14270           ASSERT(*busyrgnsp <= maxids);
14271           goto rfound;

14273 fail:
14274           ASSERT(new_rgnp != NULL);
14275           kmem_cache_free(region_cache, new_rgnp);
14276           return (HAT_INVALID_REGION_COOKIE);
14277 }

14279 /*
14280  * This function implements the shared context functionality required
14281  * when detaching a segment from an address space. It must be called
14282  * from hat_unshare() for all D(ISM) segments and from segvn_unmap(),
14283  * for segments with a valid region_cookie.
14284  * It will also be called from all seg_vn routines which change a
14285  * segment's attributes such as segvn_setprot(), segvn_setpagesize(),
14286  * segvn_clrszc() & segvn_advise(), as well as in the case of COW fault
14287  * from segvn_fault().
14288  */
14289 void
14290 hat_leave_region(struct hat *sfmmup, hat_region_cookie_t rcookie, uint_t flags)
14291 {
14292           sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14293           sf_scd_t *scdp;
14294           uint_t rhash;
14295           uint_t rid = (uint_t)((uint64_t)rcookie);
14296           hatlock_t *hatlockp = NULL;
14297           sf_region_t *rgnp;
14298           sf_region_t **prev_rgnpp;
14299           sf_region_t *cur_rgnp;
14300           void *r_obj;
14301           int i;
14302           caddr_t r_saddr;
```

```
14303         caddr_t r_eaddr;
14304         size_t  r_size;
14305         uchar_t r_pgszc;
14306         uchar_t r_type = flags & HAT_REGION_TYPE_MASK;

14308         ASSERT(sfmmup != ksfmmup);
14309         ASSERT(srdp != NULL);
14310         ASSERT(srdp->srd_refcnt > 0);
14311         ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
14312         ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
14313         ASSERT(!sfmmup->sfmmu_free || sfmmup->sfmmu_scdp == NULL);

14315         r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
14316             SFMMU_REGION_HME;

14318         if (r_type == SFMMU_REGION_ISM) {
14319                 ASSERT(SFMMU_IS_ISMRID_VALID(rid));
14320                 ASSERT(rid < SFMMU_MAX_ISM_REGIONS);
14321                 rgnp = srdp->srd_ismrgnp[rid];
14322         } else {
14323                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
14324                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
14325                 rgnp = srdp->srd_hmergnp[rid];
14326         }
14327         ASSERT(rgnp != NULL);
14328         ASSERT(rgnp->rgn_id == rid);
14329         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14330         ASSERT(!(rgnp->rgn_flags & SFMMU_REGION_FREE));
14331         ASSERT(AS_LOCK_HELD(sfmmup->sfmmu_as));
14337         ASSERT(AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

14333         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
14334         if (r_type == SFMMU_REGION_HME && sfmmup->sfmmu_as->a_xhat != NULL) {
14335                 xhat_unload_callback_all(sfmmup->sfmmu_as, rgnp->rgn_saddr,
14336                     rgnp->rgn_size, 0, NULL);
14337         }

14339         if (sfmmup->sfmmu_free) {
14340                 ulong_t rttecnt;
14341                 r_pgszc = rgnp->rgn_pgszc;
14342                 r_size = rgnp->rgn_size;

14344                 ASSERT(sfmmup->sfmmu_scdp == NULL);
14345                 if (r_type == SFMMU_REGION_ISM) {
14346                         SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14347                 } else {
14348                         /* update shme rgns ttecnt in sfmmu_ttecnt */
14349                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14350                         ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);

14352                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
14353                             -rttecnt);

14355                         SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);
14356                 }
14357         } else if (r_type == SFMMU_REGION_ISM) {
14358                 hatlockp = sfmmu_hat_enter(sfmmup);
14359                 ASSERT(rid < srdp->srd_next_ismrid);
14360                 SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14361                 scdp = sfmmup->sfmmu_scdp;
14362                 if (scdp != NULL &&
14363                     SF_RGNMAP_TEST(scdp->scd_ismregion_map, rid)) {
14364                         sfmmu_leave_scd(sfmmup, r_type);
14365                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14366                 }
14367                 sfmmu_hat_exit(hatlockp);
```



```
14368         } else {
14369                 ulong_t rttecnt;
14370                 r_pgszc = rgnp->rgn_pgszc;
14371                 r_saddr = rgnp->rgn_saddr;
14372                 r_size = rgnp->rgn_size;
14373                 r_eaddr = r_saddr + r_size;

14375                 ASSERT(r_type == SFMMU_REGION_HME);
14376                 hatlockp = sfmmu_hat_enter(sfmmup);
14377                 ASSERT(rid < srdp->srd_next_hmerid);
14378                 SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);

14380                 /*
14381                  * If region is part of an SCD call sfmmu_leave_scd().
14382                  * Otherwise if process is not exiting and has valid context
14383                  * just drop the context on the floor to lose stale TLB
14384                  * entries and force the update of tsb miss area to reflect
14385                  * the new region map. After that clean our TSB entries.
14386                  */
14387                 scdp = sfmmup->sfmmu_scdp;
14388                 if (scdp != NULL &&
14389                     SF_RGNMAP_TEST(scdp->scd_hmeregion_map, rid)) {
14390                         sfmmu_leave_scd(sfmmup, r_type);
14391                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14392                 }
14393                 sfmmu_invalidate_ctx(sfmmup);

14395                 i = TTE8K;
14396                 while (i < mmu_page_sizes) {
14397                         if (rgnp->rgn_ttecnt[i] != 0) {
14398                                 sfmmu_unload_tsb_range(sfmmup, r_saddr,
14399                                     r_eaddr, i);
14400                                 if (i < TTE4M) {
14401                                         i = TTE4M;
14402                                         continue;
14403                                 } else {
14404                                         break;
14405                                 }
14406                         }
14407                         i++;
14408                 }
14409                 /* Remove the preallocated 1/4 8k ttecnt for 4M regions. */
14410                 if (r_pgszc >= TTE4M) {
14411                         rttecnt = r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14412                         ASSERT(sfmmup->sfmmu_tsb0_4minflcnt >=
14413                             rttecnt);
14414                         sfmmup->sfmmu_tsb0_4minflcnt -= rttecnt;
14415                 }

14417                 /* update shme rgns ttecnt in sfmmu_ttecnt */
14418                 rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14419                 ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);
14420                 atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc], -rttecnt);

14422                 sfmmu_hat_exit(hatlockp);
14423                 if (scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
14424                         /* sfmmup left the scd, grow private tsb */
14425                         sfmmu_check_page_sizes(sfmmup, 1);
14426                 } else {
14427                         sfmmu_check_page_sizes(sfmmup, 0);
14428                 }
14429         }

14431         if (r_type == SFMMU_REGION_HME) {
14432                 sfmmu_unlink_from_hmeregion(sfmmup, rgnp);
14433         }
```

```
14435         r_obj = rgnp->rgn_obj;
14436         if (atomic_dec_32_nv((volatile uint_t *)&rgnp->rgn_refcnt)) {
14437                 return;
14438         }

14440         /*
14441          * looks like nobody uses this region anymore. Free it.
14442          */
14443         rhash = RGN_HASH_FUNCTION(r_obj);
14444         mutex_enter(&srdp->srd_mutex);
14445         for (prev_rgnpp = &srdp->srd_rgnhash[rhash];
14446             (cur_rgnp = *prev_rgnpp) != NULL;
14447             prev_rgnpp = &cur_rgnp->rgn_hash) {
14448                 if (cur_rgnp == rgnp && cur_rgnp->rgn_refcnt == 0) {
14449                         break;
14450                 }
14451         }

14453         if (cur_rgnp == NULL) {
14454                 mutex_exit(&srdp->srd_mutex);
14455                 return;
14456         }

14458         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14459         *prev_rgnpp = rgnp->rgn_hash;
14460         if (r_type == SFMMU_REGION_ISM) {
14461                 rgnp->rgn_flags |= SFMMU_REGION_FREE;
14462                 ASSERT(rid < srdp->srd_next_ismrid);
14463                 rgnp->rgn_next = srdp->srd_ismrgnfree;
14464                 srdp->srd_ismrgnfree = rgnp;
14465                 ASSERT(srdp->srd_ismbusyrgns > 0);
14466                 srdp->srd_ismbusyrgns--;
14467                 mutex_exit(&srdp->srd_mutex);
14468                 return;
14469         }
14470         mutex_exit(&srdp->srd_mutex);

14472         /*
14473          * Destroy region's hmeblks.
14474          */
14475         sfmmu_unload_hmeregion(srdp, rgnp);

14477         rgnp->rgn_hmeflags = 0;

14479         ASSERT(rgnp->rgn_sfmmu_head == NULL);
14480         ASSERT(rgnp->rgn_id == rid);
14481         for (i = 0; i < MMU_PAGE_SIZES; i++) {
14482                 rgnp->rgn_ttecnt[i] = 0;
14483         }
14484         rgnp->rgn_flags |= SFMMU_REGION_FREE;
14485         mutex_enter(&srdp->srd_mutex);
14486         ASSERT(rid < srdp->srd_next_hmerid);
14487         rgnp->rgn_next = srdp->srd_hmergnfree;
14488         srdp->srd_hmergnfree = rgnp;
14489         ASSERT(srdp->srd_hmebusyrgns > 0);
14490         srdp->srd_hmebusyrgns--;
14491         mutex_exit(&srdp->srd_mutex);
14492 }
```

*unchanged_portion_omitted_*

```
15109 /*
15110  * The first phase of a process joining an SCD. The hat structure is
15111  * linked to the SCD queue and then the HAT_JOIN_SCD sfmmu flag is set
15112  * and a cross-call with context invalidation is used to cause the
15113  * remaining work to be carried out in the sfmmu_tsbmiss_exception()
```



```
15114  * routine.
15115  */
15116 static void
15117 sfmmu_join_scd(sf_scd_t *scdp, sfmmu_t *sfmmup)
15118 {
15119         hatlock_t *hatlockp;
15120         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
15121         int i;
15122         sf_scd_t *old_scdp;

15124         ASSERT(srdp != NULL);
15125         ASSERT(scdp != NULL);
15126         ASSERT(scdp->scd_refcnt > 0);
15127         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as));
15133         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

15129         if ((old_scdp = sfmmup->sfmmu_scdp) != NULL) {
15130                 ASSERT(old_scdp != scdp);

15132                 mutex_enter(&old_scdp->scd_mutex);
15133                 sfmmu_from_scd_list(&old_scdp->scd_sf_list, sfmmup);
15134                 mutex_exit(&old_scdp->scd_mutex);
15135                 /*
15136                  * sfmmup leaves the old scd. Update sfmmu_ttecnt to
15137                  * include the shme rgn ttecnt for rgns that
15138                  * were in the old SCD
15139                  */
15140                 for (i = 0; i < mmu_page_sizes; i++) {
15141                         ASSERT(sfmmup->sfmmu_scdrttecnt[i] ==
15142                             old_scdp->scd_rttecnt[i]);
15143                         atomic_add_long(&sfmmup->sfmmu_ttecnt[i],
15144                             sfmmup->sfmmu_scdrttecnt[i]);
15145                 }
15146         }

15148         /*
15149          * Move sfmmu to the scd lists.
15150          */
15151         mutex_enter(&scdp->scd_mutex);
15152         sfmmu_to_scd_list(&scdp->scd_sf_list, sfmmup);
15153         mutex_exit(&scdp->scd_mutex);
15154         SF_SCD_INCR_REF(scdp);

15156         hatlockp = sfmmu_hat_enter(sfmmup);
15157         /*
15158          * For a multi-thread process, we must stop
15159          * all the other threads before joining the scd.
15160          */

15162         SFMMU_FLAGS_SET(sfmmup, HAT_JOIN_SCD);

15164         sfmmu_invalidate_ctx(sfmmup);
15165         sfmmup->sfmmu_scdp = scdp;

15167         /*
15168          * Copy scd_rttecnt into sfmmup's sfmmu_scdrttecnt, and update
15169          * sfmmu_ttecnt to not include the rgn ttecnt just joined in SCD.
15170          */
15171         for (i = 0; i < mmu_page_sizes; i++) {
15172                 sfmmup->sfmmu_scdrttecnt[i] = scdp->scd_rttecnt[i];
15173                 ASSERT(sfmmup->sfmmu_ttecnt[i] >= scdp->scd_rttecnt[i]);
15174                 atomic_add_long(&sfmmup->sfmmu_ttecnt[i],
15175                     -sfmmup->sfmmu_scdrttecnt[i]);
15176         }
15177         /* update tsb0 inflation count */
15178         if (old_scdp != NULL) {
```

```
15179                 sfmmup->sfmmu_tsb0_4minflcnt +=
15180                     old_scdp->scd_sfmmup->sfmmu_tsb0_4minflcnt;
15181         }
15182         ASSERT(sfmmup->sfmmu_tsb0_4minflcnt >=
15183             scdp->scd_sfmmup->sfmmu_tsb0_4minflcnt);
15184         sfmmup->sfmmu_tsb0_4minflcnt -= scdp->scd_sfmmup->sfmmu_tsb0_4minflcnt;

15186         sfmmu_hat_exit(hatlockp);

15188         if (old_scdp != NULL) {
15189                 SF_SCD_DECR_REF(srdp, old_scdp);
15190         }

15192 }
______unchanged_portion_omitted_

15227 /*
15228  * This routine is called in order to check if there is an SCD which matches
15229  * the process's region map if not then a new SCD may be created.
15230  */
15231 static void
15232 sfmmu_find_scd(sfmmu_t *sfmmup)
15233 {
15234         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
15235         sf_scd_t *scdp, *new_scdp;
15236         int ret;

15238         ASSERT(srdp != NULL);
15239         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as));
15245         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

15241         mutex_enter(&srdp->srd_scd_mutex);
15242         for (scdp = srdp->srd_scdp; scdp != NULL;
15243             scdp = scdp->scd_next) {
15244                 SF_RGNMAP_EQUAL(&scdp->scd_region_map,
15245                     &sfmmup->sfmmu_region_map, ret);
15246                 if (ret == 1) {
15247                         SF_SCD_INCR_REF(scdp);
15248                         mutex_exit(&srdp->srd_scd_mutex);
15249                         sfmmu_join_scd(scdp, sfmmup);
15250                         ASSERT(scdp->scd_refcnt >= 2);
15251                         atomic_dec_32((volatile uint32_t *)&scdp->scd_refcnt);
15252                         return;
15253                 } else {
15254                         /*
15255                          * If the sfmmu region map is a subset of the scd
15256                          * region map, then the assumption is that this process
15257                          * will continue attaching to ISM segments until the
15258                          * region maps are equal.
15259                          */
15260                         SF_RGNMAP_IS_SUBSET(&scdp->scd_region_map,
15261                             &sfmmup->sfmmu_region_map, ret);
15262                         if (ret == 1) {
15263                                 mutex_exit(&srdp->srd_scd_mutex);
15264                                 return;
15265                         }
15266                 }
15267         }

15269         ASSERT(scdp == NULL);
15270         /*
15271          * No matching SCD has been found, create a new one.
15272          */
15273         if ((new_scdp = sfmmu_alloc_scd(srdp, &sfmmup->sfmmu_region_map)) ==
15274             NULL) {
15275                 mutex_exit(&srdp->srd_scd_mutex);
```



```
15276                 return;
15277         }

15279         /*
15280          * sfmmu_alloc_scd() returns with a ref count of 1 on the scd.
15281          */

15283         /* Set scd_rttecnt for shme rgns in SCD */
15284         sfmmu_set_scd_rttecnt(srdp, new_scdp);

15286         /*
15287          * Link scd onto srd_scdp list and scd sfmmu onto region/iment lists.
15288          */
15289         sfmmu_link_scd_to_regions(srdp, new_scdp);
15290         sfmmu_add_scd(&srdp->srd_scdp, new_scdp);
15291         SFMMU_STAT_ADD(sf_create_scd, 1);

15293         mutex_exit(&srdp->srd_scd_mutex);
15294         sfmmu_join_scd(new_scdp, sfmmup);
15295         ASSERT(new_scdp->scd_refcnt >= 2);
15296         atomic_dec_32((volatile uint32_t *)&new_scdp->scd_refcnt);
15297 }

15299 /*
15300  * This routine is called by a process to remove itself from an SCD. It is
15301  * either called when the processes has detached from a segment or from
15302  * hat_free_start() as a result of calling exit.
15303  */
15304 static void
15305 sfmmu_leave_scd(sfmmu_t *sfmmup, uchar_t r_type)
15306 {
15307         sf_scd_t *scdp = sfmmup->sfmmu_scdp;
15308         sf_srd_t *srdp =  sfmmup->sfmmu_srdp;
15309         hatlock_t *hatlockp = TSB_HASH(sfmmup);
15310         int i;

15312         ASSERT(scdp != NULL);
15313         ASSERT(srdp != NULL);

15315         if (sfmmup->sfmmu_free) {
15316                 /*
15317                  * If the process is part of an SCD the sfmmu is unlinked
15318                  * from scd_sf_list.
15319                  */
15320                 mutex_enter(&scdp->scd_mutex);
15321                 sfmmu_from_scd_list(&scdp->scd_sf_list, sfmmup);
15322                 mutex_exit(&scdp->scd_mutex);
15323                 /*
15324                  * Update sfmmu_ttecnt to include the rgn ttecnt for rgns that
15325                  * are about to leave the SCD
15326                  */
15327                 for (i = 0; i < mmu_page_sizes; i++) {
15328                         ASSERT(sfmmup->sfmmu_scdrttecnt[i] ==
15329                             scdp->scd_rttecnt[i]);
15330                         atomic_add_long(&sfmmup->sfmmu_ttecnt[i],
15331                             sfmmup->sfmmu_scdrttecnt[i]);
15332                         sfmmup->sfmmu_scdrttecnt[i] = 0;
15333                 }
15334                 sfmmup->sfmmu_scdp = NULL;

15336                 SF_SCD_DECR_REF(srdp, scdp);
15337                 return;
15338         }

15340         ASSERT(r_type != SFMMU_REGION_ISM ||
15341             SFMMU_FLAGS_ISSET(sfmmup, HAT_ISMBUSY));
```

```
15342         ASSERT(scdp->scd_refcnt);
15343         ASSERT(!sfmmup->sfmmu_free);
15344         ASSERT(sfmmu_hat_lock_held(sfmmup));
15345         ASSERT(AS_LOCK_HELD(sfmmup->sfmmu_as));
15351         ASSERT(AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

15347         /*
15348          * Wait for ISM maps to be updated.
15349          */
15350         if (r_type != SFMMU_REGION_ISM) {
15351                 while (SFMMU_FLAGS_ISSET(sfmmup, HAT_ISMBUSY) &&
15352                     sfmmup->sfmmu_scdp != NULL) {
15353                         cv_wait(&sfmmup->sfmmu_tsb_cv,
15354                             HATLOCK_MUTEXP(hatlockp));
15355                 }

15357                 if (sfmmup->sfmmu_scdp == NULL) {
15358                         sfmmu_hat_exit(hatlockp);
15359                         return;
15360                 }
15361                 SFMMU_FLAGS_SET(sfmmup, HAT_ISMBUSY);
15362         }

15364         if (SFMMU_FLAGS_ISSET(sfmmup, HAT_JOIN_SCD)) {
15365                 SFMMU_FLAGS_CLEAR(sfmmup, HAT_JOIN_SCD);
15366                 /*
15367                  * Since HAT_JOIN_SCD was set our context
15368                  * is still invalid.
15369                  */
15370         } else {
15371                 /*
15372                  * For a multi-thread process, we must stop
15373                  * all the other threads before leaving the scd.
15374                  */

15376                 sfmmu_invalidate_ctx(sfmmup);
15377         }

15379         /* Clear all the rid's for ISM, delete flags, etc */
15380         ASSERT(SFMMU_FLAGS_ISSET(sfmmup, HAT_ISMBUSY));
15381         sfmmu_ism_hatflags(sfmmup, 0);

15383         /*
15384          * Update sfmmu_ttecnt to include the rgn ttecnt for rgns that
15385          * are in SCD before this sfmmup leaves the SCD.
15386          */
15387         for (i = 0; i < mmu_page_sizes; i++) {
15388                 ASSERT(sfmmup->sfmmu_scdrttecnt[i] ==
15389                     scdp->scd_rttecnt[i]);
15390                 atomic_add_long(&sfmmup->sfmmu_ttecnt[i],
15391                     sfmmup->sfmmu_scdrttecnt[i]);
15392                 sfmmup->sfmmu_scdrttecnt[i] = 0;
15393                 /* update ismttecnt to include SCD ism before hat leaves SCD */
15394                 sfmmup->sfmmu_ismttecnt[i] += sfmmup->sfmmu_scdismttecnt[i];
15395                 sfmmup->sfmmu_scdismttecnt[i] = 0;
15396         }
15397         /* update tsb0 inflation count */
15398         sfmmup->sfmmu_tsb0_4minflcnt += scdp->scd_sfmmup->sfmmu_tsb0_4minflcnt;

15400         if (r_type != SFMMU_REGION_ISM) {
15401                 SFMMU_FLAGS_CLEAR(sfmmup, HAT_ISMBUSY);
15402         }
15403         sfmmup->sfmmu_scdp = NULL;

15405         sfmmu_hat_exit(hatlockp);
```



```
15407         /*
15408          * Unlink sfmmu from scd_sf_list this can be done without holding
15409          * the hat lock as we hold the sfmmu_as lock which prevents
15410          * hat_join_region from adding this thread to the scd again. Other
15411          * threads check if sfmmu_scdp is NULL under hat lock and if it's NULL
15412          * they won't get here, since sfmmu_leave_scd() clears sfmmu_scdp
15413          * while holding the hat lock.
15414          */
15415         mutex_enter(&scdp->scd_mutex);
15416         sfmmu_from_scd_list(&scdp->scd_sf_list, sfmmup);
15417         mutex_exit(&scdp->scd_mutex);
15418         SFMMU_STAT(sf_leave_scd);

15420         SF_SCD_DECR_REF(srdp, scdp);
15421         hatlockp = sfmmu_hat_enter(sfmmup);

15423 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   38360 Wed Nov 25 13:59:42 2015
new/usr/src/uts/sparc/v9/os/simulator.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_

 841 /*
 842  * simulate unimplemented instructions (popc, ldqf{a}, stqf{a})
 843  */
 844 int
 845 simulate_unimp(struct regs *rp, caddr_t *badaddr)
 846 {
 847         uint_t  inst, optype, op3, asi;
 848         uint_t  rs1, rd;
 849         uint_t  ignor, i;
 850         machpcb_t *mpcb = lwptompcb(ttolwp(curthread));
 851         int     nomatch = 0;
 852         caddr_t addr = (caddr_t)rp->r_pc;
 853         struct as *as;
 854         caddr_t ka;
 855         pfn_t   pfnum;
 856         page_t *pp;
 857         proc_t *p = ttoproc(curthread);
 858         struct seg *mapseg;
 859         struct segvn_data *svd;

 861         ASSERT(USERMODE(rp->r_tstate));
 862         inst = fetch_user_instr(addr);
 863         if (inst == (uint_t)-1) {
 864                 mpcb->mpcb_illexcaddr = addr;
 865                 mpcb->mpcb_illexcinsn = (uint32_t)-1;
 866                 return (SIMU_ILLEGAL);
 867         }

 869         /*
 870          * When fixing dirty v8 instructions there's a race if two processors
 871          * are executing the dirty executable at the same time.  If one
 872          * cleans the instruction as the other is executing it the second
 873          * processor will see a clean instruction when it comes through this
 874          * code and will return SIMU_ILLEGAL.  To work around the race
 875          * this code will keep track of the last illegal instruction seen
 876          * by each lwp and will only take action if the illegal instruction
 877          * is repeatable.
 878          */
 879         if (addr != mpcb->mpcb_illexcaddr ||
 880             inst != mpcb->mpcb_illexcinsn)
 881                 nomatch = 1;
 882         mpcb->mpcb_illexcaddr = addr;
 883         mpcb->mpcb_illexcinsn = inst;

 885         /* instruction fields */
 886         i = (inst >> 13) & 0x1;
 887         rd = (inst >> 25) & 0x1f;
 888         optype = (inst >> 30) & 0x3;
 889         op3 = (inst >> 19) & 0x3f;
 890         ignor = (inst >> 5) & 0xff;
 891         if (IS_IBIT_SET(inst)) {
 892                 asi = (uint32_t)((rp->r_tstate >> TSTATE_ASI_SHIFT) &
 893                     TSTATE_ASI_MASK);
 894         } else {
 895                 asi = ignor;
 896         }

 898         if (IS_VIS1(optype, op3) ||
 899             IS_PARTIAL_OR_SHORT_FLOAT_LD_ST(optype, op3, asi) ||
```



```
 900             IS_FLOAT_QUAD_OP(optype, op3)) {
 901                 klwp_t *lwp = ttolwp(curthread);
 902                 kfpu_t *fp = lwptofpu(lwp);
 903                 if (fpu_exists) {
 904                         if (!(_fp_read_fprs() & FPRS_FEF))
 905                                 fp_enable();
 906                         _fp_read_pfsr(&fp->fpu_fsr);
 907                 } else {
 908                         if (!fp->fpu_en)
 909                                 fp_enable();
 910                 }
 911                 fp_precise(rp);
 912                 return (SIMU_RETRY);
 913         }

 915         if (optype == 2 && op3 == IOP_V8_POPC) {
 916                 return (simulate_popc(rp, badaddr, inst));
 917         } else if (optype == 3 && op3 == IOP_V8_POPC) {
 918                 return (SIMU_ILLEGAL);
 919         } else if (optype == OP_V8_ARITH && op3 == IOP_V8_MULScc) {
 920                 return (simulate_mulscc(rp, badaddr, inst));
 921         }

 923         if (optype == OP_V8_LDSTR) {
 924                 if (op3 == IOP_V8_LDQF || op3 == IOP_V8_LDQFA ||
 925                     op3 == IOP_V8_STQF || op3 == IOP_V8_STQFA)
 926                         return (do_unaligned(rp, badaddr));
 927         }

 929         /* This is a new instruction so illexccnt should also be set. */
 930         if (nomatch) {
 931                 mpcb->mpcb_illexccnt = 0;
 932                 return (SIMU_RETRY);
 933         }

 935         /*
 936          * In order to keep us from entering into an infinite loop while
 937          * attempting to clean up faulty instructions, we will return
 938          * SIMU_ILLEGAL once we've cleaned up the instruction as much
 939          * as we can, and still end up here.
 940          */
 941         if (mpcb->mpcb_illexccnt >= 3)
 942                 return (SIMU_ILLEGAL);

 944         mpcb->mpcb_illexccnt += 1;

 946         /*
 947          * The rest of the code handles v8 binaries with instructions
 948          * that have dirty (non-zero) bits in reserved or 'ignored'
 949          * fields; these will cause core dumps on v9 machines.
 950          *
 951          * We only clean dirty instructions in 32-bit programs (ie, v8)
 952          * running on SPARCv9 processors.  True v9 programs are forced
 953          * to use the instruction set as intended.
 954          */
 955         if (lwp_getdatamodel(curthread->t_lwp) != DATAMODEL_ILP32)
 956                 return (SIMU_ILLEGAL);
 957         switch (optype) {
 958         case OP_V8_BRANCH:
 959         case OP_V8_CALL:
 960                 return (SIMU_ILLEGAL);  /* these don't have ignored fields */
 961                 /*NOTREACHED*/
 962         case OP_V8_ARITH:
 963                 switch (op3) {
 964                 case IOP_V8_RETT:
 965                         if (rd == 0 && !(i == 0 && ignor))
```

```
966                             return (SIMU_ILLEGAL);
967                     if (rd)
968                             inst &= ~(0x1f << 25);
969                     if (i == 0 && ignor)
970                             inst &= ~(0xff << 5);
971                     break;
972             case IOP_V8_TCC:
973                     if (i == 0 && ignor != 0) {
974                             inst &= ~(0xff << 5);
975                     } else if (i == 1 && (((inst >> 7) & 0x3f) != 0)) {
976                             inst &= ~(0x3f << 7);
977                     } else {
978                             return (SIMU_ILLEGAL);
979                     }
980                     break;
981             case IOP_V8_JMPL:
982             case IOP_V8_RESTORE:
983             case IOP_V8_SAVE:
984                     if ((op3 == IOP_V8_RETT && rd) ||
985                         (i == 0 && ignor)) {
986                             inst &= ~(0xff << 5);
987                     } else {
988                             return (SIMU_ILLEGAL);
989                     }
990                     break;
991             case IOP_V8_FCMP:
992                     if (rd == 0)
993                             return (SIMU_ILLEGAL);
994                     inst &= ~(0x1f << 25);
995                     break;
996             case IOP_V8_RDASR:
997                     rs1 = ((inst >> 14) & 0x1f);
998                     if (rs1 == 1 || (rs1 >= 7 && rs1 <= 14)) {
999                             /*
1000                             * The instruction specifies an invalid
1001                             * state register - better bail out than
1002                             * "fix" it when we're not sure what was
1003                             * intended.
1004                             */
1005                            return (SIMU_ILLEGAL);
1006                    }
1007                            /*
1008                             * Note: this case includes the 'stbar'
1009                             * instruction (rs1 == 15 && i == 0).
1010                             */
1011                            if ((ignor = (inst & 0x3fff)) != 0)
1012                                    inst &= ~(0x3fff);
1013                    break;
1014            case IOP_V8_SRA:
1015            case IOP_V8_SRL:
1016            case IOP_V8_SLL:
1017                    if (ignor == 0)
1018                            return (SIMU_ILLEGAL);
1019                    inst &= ~(0xff << 5);
1020                    break;
1021            case IOP_V8_ADD:
1022            case IOP_V8_AND:
1023            case IOP_V8_OR:
1024            case IOP_V8_XOR:
1025            case IOP_V8_SUB:
1026            case IOP_V8_ANDN:
1027            case IOP_V8_ORN:
1028            case IOP_V8_XNOR:
1029            case IOP_V8_ADDC:
1030            case IOP_V8_UMUL:
1031            case IOP_V8_SMUL:
```



```
1032            case IOP_V8_SUBC:
1033            case IOP_V8_UDIV:
1034            case IOP_V8_SDIV:
1035            case IOP_V8_ADDcc:
1036            case IOP_V8_ANDcc:
1037            case IOP_V8_ORcc:
1038            case IOP_V8_XORcc:
1039            case IOP_V8_SUBcc:
1040            case IOP_V8_ANDNcc:
1041            case IOP_V8_ORNcc:
1042            case IOP_V8_XNORcc:
1043            case IOP_V8_ADDCcc:
1044            case IOP_V8_UMULcc:
1045            case IOP_V8_SMULcc:
1046            case IOP_V8_SUBCcc:
1047            case IOP_V8_UDIVcc:
1048            case IOP_V8_SDIVcc:
1049            case IOP_V8_TADDcc:
1050            case IOP_V8_TSUBcc:
1051            case IOP_V8_TADDccTV:
1052            case IOP_V8_TSUBccTV:
1053            case IOP_V8_MULScc:
1054            case IOP_V8_WRASR:
1055            case IOP_V8_FLUSH:
1056                    if (i != 0 || ignor == 0)
1057                            return (SIMU_ILLEGAL);
1058                    inst &= ~(0xff << 5);
1059                    break;
1060            default:
1061                    return (SIMU_ILLEGAL);
1062            }
1063            break;
1064    case OP_V8_LDSTR:
1065            switch (op3) {
1066            case IOP_V8_STFSR:
1067            case IOP_V8_LDFSR:
1068                    if (rd == 0 && !(i == 0 && ignor))
1069                            return (SIMU_ILLEGAL);
1070                    if (rd)
1071                            inst &= ~(0x1f << 25);
1072                    if (i == 0 && ignor)
1073                            inst &= ~(0xff << 5);
1074                    break;
1075            default:
1076                    if (optype == OP_V8_LDSTR && !IS_LDST_ALT(op3) &&
1077                        i == 0 && ignor)
1078                            inst &= ~(0xff << 5);
1079                    else
1080                            return (SIMU_ILLEGAL);
1081                    break;
1082            }
1083            break;
1084    default:
1085            return (SIMU_ILLEGAL);
1086    }

1088    as = p->p_as;

1090    AS_LOCK_ENTER(as, RW_READER);
1090    AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1091    mapseg = as_findseg(as, (caddr_t)rp->r_pc, 0);
1092    ASSERT(mapseg != NULL);
1093    svd = (struct segvn_data *)mapseg->s_data;

1095    /*
1096     * We only create COW page for MAP_PRIVATE mappings.
```

```
1097          */
1098         SEGVN_LOCK_ENTER(as, &svd->lock, RW_READER);
1099         if ((svd->type & MAP_TYPE) & MAP_SHARED) {
1100                 SEGVN_LOCK_EXIT(as, &svd->lock);
1101                 AS_LOCK_EXIT(as);
1101                 AS_LOCK_EXIT(as, &as->a_lock);
1102                 return (SIMU_ILLEGAL);
1103         }
1104         SEGVN_LOCK_EXIT(as, &svd->lock);
1105         AS_LOCK_EXIT(as);
1105         AS_LOCK_EXIT(as, &as->a_lock);

1107         /*
1108          * A "flush" instruction using the user PC's vaddr will not work
1109          * here, at least on Spitfire. Instead we create a temporary kernel
1110          * mapping to the user's text page, then modify and flush that.
1111          * Break COW by locking user page.
1112          */
1113         if (as_fault(as->a_hat, as, (caddr_t)(rp->r_pc & PAGEMASK), PAGESIZE,
1114             F_SOFTLOCK, S_READ))
1115                 return (SIMU_FAULT);

1117         AS_LOCK_ENTER(as, RW_READER);
1117         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1118         pfnum = hat_getpfnum(as->a_hat, (caddr_t)rp->r_pc);
1119         AS_LOCK_EXIT(as);
1119         AS_LOCK_EXIT(as, &as->a_lock);
1120         if (pf_is_memory(pfnum)) {
1121                 pp = page_numtopp_nolock(pfnum);
1122                 ASSERT(pp == NULL || PAGE_LOCKED(pp));
1123         } else {
1124                 (void) as_fault(as->a_hat, as, (caddr_t)(rp->r_pc & PAGEMASK),
1125                     PAGESIZE, F_SOFTUNLOCK, S_READ);
1126                 return (SIMU_FAULT);
1127         }

1129         AS_LOCK_ENTER(as, RW_READER);
1129         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1130         ka = ppmapin(pp, PROT_READ|PROT_WRITE, (caddr_t)rp->r_pc);
1131         *(uint_t *)(ka + (uintptr_t)(rp->r_pc % PAGESIZE)) = inst;
1132         doflush(ka + (uintptr_t)(rp->r_pc % PAGESIZE));
1133         ppmapout(ka);
1134         AS_LOCK_EXIT(as);
1134         AS_LOCK_EXIT(as, &as->a_lock);

1136         (void) as_fault(as->a_hat, as, (caddr_t)(rp->r_pc & PAGEMASK),
1137             PAGESIZE, F_SOFTUNLOCK, S_READ);
1138         return (SIMU_RETRY);
1139 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   12249 Wed Nov 25 13:59:42 2015
new/usr/src/uts/sparc/v9/vm/seg_nf.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_


/*
 * Create a no-fault segment.
 *
 * The no-fault segment is not technically necessary, as the code in
 * nfload() in trap.c will emulate the SPARC instruction and load
 * a value of zero in the destination register.
 *
 * However, this code tries to put a page of zero's at the nofault address
 * so that subsequent non-faulting loads to the same page will not
 * trap with a tlb miss.
 *
 * In order to help limit the number of segments we merge adjacent nofault
 * segments into a single segment.  If we get a large number of segments
 * we'll also try to delete a random other nf segment.
 */
/* ARGSUSED */
int
segnf_create(struct seg *seg, void *argsp)
{
        uint_t prot;
        pgcnt_t vacpgs;
        u_offset_t off = 0;
        caddr_t vaddr = NULL;
        int i, color;
        struct seg *s1;
        struct seg *s2;
        size_t size;
        struct as *as = seg->s_as;

        ASSERT(as && AS_WRITE_HELD(as));
        ASSERT(as && AS_WRITE_HELD(as, &as->a_lock));

        /*
         * Need a page per virtual color or just 1 if no vac.
         */
        mutex_enter(&segnf_lock);
        if (nfpp == NULL) {
                struct seg kseg;

                vacpgs = 1;
                if (shm_alignment > PAGESIZE) {
                        vacpgs = shm_alignment >> PAGESHIFT;
                }

                nfpp = kmem_alloc(sizeof (*nfpp) * vacpgs, KM_SLEEP);

                kseg.s_as = &kas;
                for (i = 0; i < vacpgs; i++, off += PAGESIZE,
                    vaddr += PAGESIZE) {
                        nfpp[i] = page_create_va(&nfvp, off, PAGESIZE,
                            PG_WAIT | PG_NORELOC, &kseg, vaddr);
                        page_io_unlock(nfpp[i]);
                        page_downgrade(nfpp[i]);
                        pagezero(nfpp[i], 0, PAGESIZE);
                }
        }
        mutex_exit(&segnf_lock);
```



```
        hat_map(as->a_hat, seg->s_base, seg->s_size, HAT_MAP);

        /*
         * s_data can't be NULL because of ASSERTS in the common vm code.
         */
        seg->s_ops = &segnf_ops;
        seg->s_data = seg;
        seg->s_flags |= S_PURGE;

        mutex_enter(&as->a_contents);
        as->a_flags |= AS_NEEDSPURGE;
        mutex_exit(&as->a_contents);

        prot = PROT_READ;
        color = addr_to_vcolor(seg->s_base);
        if (as != &kas)
                prot |= PROT_USER;
        hat_memload(as->a_hat, seg->s_base, nfpp[color],
            prot | HAT_NOFAULT, HAT_LOAD);

        /*
         * At this point see if we can concatenate a segment to
         * a non-fault segment immediately before and/or after it.
         */
        if ((s1 = AS_SEGPREV(as, seg)) != NULL &&
            s1->s_ops == &segnf_ops &&
            s1->s_base + s1->s_size == seg->s_base) {
                size = s1->s_size;
                seg_free(s1);
                seg->s_base -= size;
                seg->s_size += size;
        }

        if ((s2 = AS_SEGNEXT(as, seg)) != NULL &&
            s2->s_ops == &segnf_ops &&
            seg->s_base + seg->s_size == s2->s_base) {
                size = s2->s_size;
                seg_free(s2);
                seg->s_size += size;
        }

        /*
         * if we already have a lot of segments, try to delete some other
         * nofault segment to reduce the probability of uncontrolled segment
         * creation.
         *
         * the code looks around quickly (no more than MAXNFSEARCH segments
         * each way) for another NF segment and then deletes it.
         */
        if (avl_numnodes(&as->a_segtree) > MAXSEGFORNF) {
                size = 0;
                s2 = NULL;
                s1 = AS_SEGPREV(as, seg);
                while (size++ < MAXNFSEARCH && s1 != NULL) {
                        if (s1->s_ops == &segnf_ops)
                                s2 = s1;
                        s1 = AS_SEGPREV(s1->s_as, seg);
                }
                if (s2 == NULL) {
                        s1 = AS_SEGNEXT(as, seg);
                        while (size-- > 0 && s1 != NULL) {
                                if (s1->s_ops == &segnf_ops)
                                        s2 = s1;
                                s1 = AS_SEGNEXT(as, seg);
                        }
                }
```

```
285                 if (s2 != NULL)
286                         seg_unmap(s2);
287         }

289         return (0);
290 }
_____unchanged_portion_omitted_

303 /*
304  * Split a segment at addr for length len.
305  */
306 static int
307 segnf_unmap(struct seg *seg, caddr_t addr, size_t len)
308 {
309         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
309         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

311         /*
312          * Check for bad sizes.
313          */
314         if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
315             (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET)) {
316                 cmn_err(CE_PANIC, "segnf_unmap: bad unmap size");
317         }

319         /*
320          * Unload any hardware translations in the range to be taken out.
321          */
322         hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD_UNMAP);

324         if (addr == seg->s_base && len == seg->s_size) {
325                 /*
326                  * Freeing entire segment.
327                  */
328                 seg_free(seg);
329         } else if (addr == seg->s_base) {
330                 /*
331                  * Freeing the beginning of the segment.
332                  */
333                 seg->s_base += len;
334                 seg->s_size -= len;
335         } else if (addr + len == seg->s_base + seg->s_size) {
336                 /*
337                  * Freeing the end of the segment.
338                  */
339                 seg->s_size -= len;
340         } else {
341                 /*
342                  * The section to go is in the middle of the segment, so we
343                  * have to cut it into two segments.  We shrink the existing
344                  * "seg" at the low end, and create "nseg" for the high end.
345                  */
346                 caddr_t nbase = addr + len;
347                 size_t nsize = (seg->s_base + seg->s_size) - nbase;
348                 struct seg *nseg;

350                 /*
351                  * Trim down "seg" before trying to stick "nseg" into the as.
352                  */
353                 seg->s_size = addr - seg->s_base;
354                 nseg = seg_alloc(seg->s_as, nbase, nsize);
355                 if (nseg == NULL)
356                         cmn_err(CE_PANIC, "segnf_unmap: seg_alloc failed");

358                 /*
359                  * s_data can't be NULL because of ASSERTs in common VM code.
```



```
360                  */
361                 nseg->s_ops = seg->s_ops;
362                 nseg->s_data = nseg;
363                 nseg->s_flags |= S_PURGE;
364                 mutex_enter(&seg->s_as->a_contents);
365                 seg->s_as->a_flags |= AS_NEEDSPURGE;
366                 mutex_exit(&seg->s_as->a_contents);
367         }

369         return (0);
370 }

372 /*
373  * Free a segment.
374  */
375 static void
376 segnf_free(struct seg *seg)
377 {
378         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as));
378         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
379 }
_____unchanged_portion_omitted_

390 /* ARGSUSED */
391 static int
392 segnf_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
393 {
394         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
394         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
395         return (EACCES);
396 }

398 /* ARGSUSED */
399 static int
400 segnf_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
401 {
402         uint_t sprot;
403         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
403         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

405         sprot = seg->s_as == &kas ?  PROT_READ : PROT_READ|PROT_USER;
406         return ((prot & sprot) == prot ? 0 : EACCES);
407 }
_____unchanged_portion_omitted_

422 static int
423 segnf_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
424 {
425         size_t pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;
426         size_t p;
427         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
427         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

429         for (p = 0; p < pgno; ++p)
430                 protv[p] = PROT_READ;
431         return (0);
432 }

434 /* ARGSUSED */
435 static u_offset_t
436 segnf_getoffset(struct seg *seg, caddr_t addr)
437 {
438         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
438         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

440         return ((u_offset_t)0);
```

```
 441 }

 443 /* ARGSUSED */
 444 static int
 445 segnf_gettype(struct seg *seg, caddr_t addr)
 446 {
 447         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 447         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

 449         return (MAP_SHARED);
 450 }

 452 /* ARGSUSED */
 453 static int
 454 segnf_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
 455 {
 456         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
 456         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

 458         *vpp = &nfvp;
 459         return (0);
 460 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   23580 Wed Nov 25 13:59:43 2015
new/usr/src/uts/sun4/io/rootnex.c
patch as-lock-macro-simplification
**********************************************************
______unchanged_portion_omitted_


/*
 * Shorthand defines
 */

#define DMAOBJ_PP_PP    dmao_obj.pp_obj.pp_pp
#define DMAOBJ_PP_OFF   dmao_ogj.pp_obj.pp_offset
#define ALO             dma_lim->dlim_addr_lo
#define AHI             dma_lim->dlim_addr_hi
#define OBJSIZE         dmareq->dmar_object.dmao_size
#define ORIGVADDR       dmareq->dmar_object.dmao_obj.virt_obj.v_addr
#define RED             ((mp->dmai_rflags & DDI_DMA_REDZONE)? 1 : 0)
#define DIRECTION       (mp->dmai_rflags & DDI_DMA_RDWR)

/*
 * rootnex_map_fault:
 *
 *      fault in mappings for requestors
 */

/*ARGSUSED*/
static int
rootnex_map_fault(dev_info_t *dip, dev_info_t *rdip,
    struct hat *hat, struct seg *seg, caddr_t addr,
    struct devpage *dp, pfn_t pfn, uint_t prot, uint_t lock)
{
        extern struct seg_ops segdev_ops;

        DPRINTF(ROOTNEX_MAP_DEBUG, ("rootnex_map_fault: address <%p> "
            "pfn <%lx>", (void *)addr, pfn));
        DPRINTF(ROOTNEX_MAP_DEBUG, (" Seg <%s>\n",
            seg->s_ops == &segdev_ops ? "segdev" :
            seg == &kvseg ? "segkmem" : "NONE!"));

        /*
         * This is all terribly broken, but it is a start
         *
         * XXX  Note that this test means that segdev_ops
         *      must be exported from seg_dev.c.
         * XXX  What about devices with their own segment drivers?
         */
        if (seg->s_ops == &segdev_ops) {
                register struct segdev_data *sdp =
                    (struct segdev_data *)seg->s_data;

                if (hat == NULL) {
                        /*
                         * This is one plausible interpretation of
                         * a null hat i.e. use the first hat on the
                         * address space hat list which by convention is
                         * the hat of the system MMU.  At alternative
                         * would be to panic .. this might well be better ..
                         */
                        ASSERT(AS_READ_HELD(seg->s_as));
                        ASSERT(AS_READ_HELD(seg->s_as, &seg->s_as->a_lock));
                        hat = seg->s_as->a_hat;
                        cmn_err(CE_NOTE, "rootnex_map_fault: nil hat");
                }
                hat_devload(hat, addr, MMU_PAGESIZE, pfn, prot | sdp->hat_attr,
```



```
                    (lock ? HAT_LOAD_LOCK : HAT_LOAD));
        } else if (seg == &kvseg && dp == (struct devpage *)0) {
                hat_devload(kas.a_hat, addr, MMU_PAGESIZE, pfn, prot,
                    HAT_LOAD_LOCK);
        } else
                return (DDI_FAILURE);
        return (DDI_SUCCESS);
}
______unchanged_portion_omitted_
```