```
**********************************************************
   21346 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/findstack.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*ARGSUSED*/
int
stacks(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        size_t idx;

        char *seen = NULL;

        const char *caller_str = NULL;
        const char *excl_caller_str = NULL;
        uintptr_t caller = 0, excl_caller = 0;
        const char *module_str = NULL;
        const char *excl_module_str = NULL;
        stacks_module_t module, excl_module;
        const char *sobj = NULL;
        const char *excl_sobj = NULL;
        uintptr_t sobj_ops = 0, excl_sobj_ops = 0;
        const char *tstate_str = NULL;
        const char *excl_tstate_str = NULL;
        uint_t tstate = -1U;
        uint_t excl_tstate = -1U;
        uint_t printed = 0;

        uint_t all = 0;
        uint_t force = 0;
        uint_t interesting = 0;
        uint_t verbose = 0;

        /*
         * We have a slight behavior difference between having piped
         * input and 'addr::stacks'.  Without a pipe, we assume the
         * thread pointer given is a representative thread, and so
         * we include all similar threads in the system in our output.
         *
         * With a pipe, we filter down to just the threads in our
         * input.
         */
        uint_t addrspec = (flags & DCMD_ADDRSPEC);
        uint_t only_matching = addrspec && (flags & DCMD_PIPE);

        mdb_pipe_t p;

        bzero(&module, sizeof (module));
        bzero(&excl_module, sizeof (excl_module));

        if (mdb_getopts(argc, argv,
            'a', MDB_OPT_SETBITS, TRUE, &all,
            'f', MDB_OPT_SETBITS, TRUE, &force,
            'i', MDB_OPT_SETBITS, TRUE, &interesting,
            'v', MDB_OPT_SETBITS, TRUE, &verbose,
            'c', MDB_OPT_STR, &caller_str,
            'C', MDB_OPT_STR, &excl_caller_str,
            'm', MDB_OPT_STR, &module_str,
            'M', MDB_OPT_STR, &excl_module_str,
            's', MDB_OPT_STR, &sobj,
            'S', MDB_OPT_STR, &excl_sobj,
            't', MDB_OPT_STR, &tstate_str,
            'T', MDB_OPT_STR, &excl_tstate_str,
            NULL) != argc)
```

```
                return (DCMD_USAGE);

        if (interesting) {
                if (sobj != NULL || excl_sobj != NULL ||
                    tstate_str != NULL || excl_tstate_str != NULL) {
                        mdb_warn(
                            "stacks: -i is incompatible with -[sStT]\n");
                        return (DCMD_USAGE);
                }
                excl_sobj = "CV";
                excl_tstate_str = "FREE";
        }

        if (caller_str != NULL) {
                mdb_set_dot(0);
                if (mdb_eval(caller_str) != 0) {
                        mdb_warn("stacks: evaluation of \"%s\" failed",
                            caller_str);
                        return (DCMD_ABORT);
                }
                caller = mdb_get_dot();
        }

        if (excl_caller_str != NULL) {
                mdb_set_dot(0);
                if (mdb_eval(excl_caller_str) != 0) {
                        mdb_warn("stacks: evaluation of \"%s\" failed",
                            excl_caller_str);
                        return (DCMD_ABORT);
                }
                excl_caller = mdb_get_dot();
        }
        mdb_set_dot(addr);

        if (module_str != NULL && stacks_module_find(module_str, &module) != 0)
                return (DCMD_ABORT);

        if (excl_module_str != NULL &&
            stacks_module_find(excl_module_str, &excl_module) != 0)
                return (DCMD_ABORT);

        if (sobj != NULL && text_to_sobj(sobj, &sobj_ops) != 0)
                return (DCMD_USAGE);

        if (excl_sobj != NULL && text_to_sobj(excl_sobj, &excl_sobj_ops) != 0)
                return (DCMD_USAGE);

        if (sobj_ops != 0 && excl_sobj_ops != 0) {
                mdb_warn("stacks: only one of -s and -S can be specified\n");
                return (DCMD_USAGE);
        }

        if (tstate_str != NULL && text_to_tstate(tstate_str, &tstate) != 0)
                return (DCMD_USAGE);

        if (excl_tstate_str != NULL &&
            text_to_tstate(excl_tstate_str, &excl_tstate) != 0)
                return (DCMD_USAGE);

        if (tstate != -1U && excl_tstate != -1U) {
                mdb_warn("stacks: only one of -t and -T can be specified\n");
                return (DCMD_USAGE);
        }

        /*
         * If there's an address specified, we're going to further filter
```

```
	 * to only entries which have an address in the input.  To reduce
	 * overhead (and make the sorted output come out right), we
	 * use mdb_get_pipe() to grab the entire pipeline of input, then
	 * use qsort() and bsearch() to speed up the search.
	 */
	if (addrspec) {
		mdb_get_pipe(&p);
		if (p.pipe_data == NULL || p.pipe_len == 0) {
			p.pipe_data = &addr;
			p.pipe_len = 1;
		}
		qsort(p.pipe_data, p.pipe_len, sizeof (uintptr_t),
		    uintptrcomp);

		/* remove any duplicates in the data */
		idx = 0;
		while (idx < p.pipe_len - 1) {
			uintptr_t *data = &p.pipe_data[idx];
			size_t len = p.pipe_len - idx;

			if (data[0] == data[1]) {
				memmove(data, data + 1,
				    (len - 1) * sizeof (*data));
				p.pipe_len--;
				continue; /* repeat without incrementing idx */
			}
			idx++;
		}

		seen = mdb_zalloc(p.pipe_len, UM_SLEEP | UM_GC);
	}

	/*
	 * Force a cleanup if we're connected to a live system. Never
	 * do a cleanup after the first invocation around the loop.
	 */
	force |= (mdb_get_state() == MDB_STATE_RUNNING);
	if (force && (flags & (DCMD_LOOPFIRST|DCMD_LOOP)) == DCMD_LOOP)
		force = 0;

	stacks_cleanup(force);

	if (stacks_state == STACKS_STATE_CLEAN) {
		int res = stacks_run(verbose, addrspec ? &p : NULL);
		if (res != DCMD_OK)
			return (res);
	}

	for (idx = 0; idx < stacks_array_size; idx++) {
		stacks_entry_t *sep = stacks_array[idx];
		stacks_entry_t *cur = sep;
		int frame;
		size_t count = sep->se_count;

		if (addrspec) {
			stacks_entry_t *head = NULL, *tail = NULL, *sp;
			size_t foundcount = 0;
			/*
			 * We use the now-unused hash chain field se_next to
			 * link together the dups which match our list.
			 */
			for (sp = sep; sp != NULL; sp = sp->se_dup) {
				uintptr_t *entry = bsearch(&sp->se_thread,
				    p.pipe_data, p.pipe_len, sizeof (uintptr_t),
				    uintptrcomp);
				if (entry != NULL) {
```

```
					foundcount++;
					seen[entry - p.pipe_data]++;
					if (head == NULL)
						head = sp;
					else
						tail->se_next = sp;
					tail = sp;
					sp->se_next = NULL;
				}
			}
			if (head == NULL)
				continue;	/* no match, skip entry */

			if (only_matching) {
				cur = sep = head;
				count = foundcount;
			}
		}

		if (caller != 0 && !stacks_has_caller(sep, caller))
			continue;

		if (excl_caller != 0 && stacks_has_caller(sep, excl_caller))
			continue;

		if (module.sm_size != 0 && !stacks_has_module(sep, &module))
			continue;

		if (excl_module.sm_size != 0 &&
		    stacks_has_module(sep, &excl_module))
			continue;

		if (tstate != -1U) {
			if (tstate == TSTATE_PANIC) {
				if (!sep->se_panic)
					continue;
			} else if (sep->se_panic || sep->se_tstate != tstate)
				continue;
		}
		if (excl_tstate != -1U) {
			if (excl_tstate == TSTATE_PANIC) {
				if (sep->se_panic)
					continue;
			} else if (!sep->se_panic &&
			    sep->se_tstate == excl_tstate)
				continue;
		}

		if (sobj_ops == SOBJ_ALL) {
			if (sep->se_sobj_ops == 0)
				continue;
		} else if (sobj_ops != 0) {
			if (sobj_ops != sep->se_sobj_ops)
				continue;
		}

		if (!(interesting && sep->se_panic)) {
			if (excl_sobj_ops == SOBJ_ALL) {
				if (sep->se_sobj_ops != 0)
					continue;
			} else if (excl_sobj_ops != 0) {
				if (excl_sobj_ops == sep->se_sobj_ops)
					continue;
			}
		}
```

```
                if (flags & DCMD_PIPE_OUT) {
                        while (sep != NULL) {
                                mdb_printf("%lr\n", sep->se_thread);
                                sep = only_matching ?
                                    sep->se_next : sep->se_dup;
                        }
                        continue;
                }

                if (all || !printed) {
                        mdb_printf("%<u>%-?s %-8s %-?s %8s%</u>\n",
                            "THREAD", "STATE", "SOBJ", "COUNT");
                        printed = 1;
                }

                do {
                        char state[20];
                        char sobj[100];

                        tstate_to_text(cur->se_tstate, cur->se_panic,
                            state, sizeof (state));
                        sobj_to_text(cur->se_sobj_ops,
                            sobj, sizeof (sobj));

                        if (cur == sep)
                                mdb_printf("%-?p %-8s %-?s %8d\n",
                                    cur->se_thread, state, sobj, count);
                        else
                                mdb_printf("%-?p %-8s %-?s %8s\n",
                                    cur->se_thread, state, sobj, "-");

                        cur = only_matching ? cur->se_next : cur->se_dup;
                } while (all && cur != NULL);

                if (sep->se_failed != 0) {
                        char *reason;
                        switch (sep->se_failed) {
                        case FSI_FAIL_NOTINMEMORY:
                                reason = "thread not in memory";
                                break;
                        case FSI_FAIL_THREADCORRUPT:
                                reason = "thread structure stack info corrupt";
                                break;
                        case FSI_FAIL_STACKNOTFOUND:
                                reason = "no consistent stack found";
                                break;
                        default:
                                reason = "unknown failure";
                                break;
                        }
                        mdb_printf("%?s <%s>\n", "", reason);
                }

                for (frame = 0; frame < sep->se_depth; frame++)
                        mdb_printf("%?s %a\n", "", sep->se_stack[frame]);
                if (sep->se_overflow)
                        mdb_printf("%?s ... truncated ...\n", "");
                mdb_printf("\n");
        }

        if (flags & DCMD_ADDRSPEC) {
                for (idx = 0; idx < p.pipe_len; idx++)
                        if (seen[idx] == 0)
                                mdb_warn("stacks: %p not in thread list\n",
                                    p.pipe_data[idx]);
        }
```

```
        return (DCMD_OK);
}
```

______unchanged_portion_omitted_

****
2801 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/findstack.h
6583 remove whole-process swapping
****
______***unchanged_portion_omitted_***

```
#define FSI_FAIL_BADTHREAD      1
#define FSI_FAIL_THREADCORRUPT  2
#define FSI_FAIL_STACKNOTFOUND  3
#define FSI_FAIL_NOTINMEMORY    2
#define FSI_FAIL_THREADCORRUPT  3
#define FSI_FAIL_STACKNOTFOUND  4

typedef struct stacks_module {
        char            sm_name[MAXPATHLEN]; /* name of module */
        uintptr_t       sm_text;        /* base address of text in module */
        size_t          sm_size;        /* size of text in module */
} stacks_module_t;
```
______***unchanged_portion_omitted_***

```
**********************************************************
   11424 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/findstack_subr.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*ARGSUSED*/
int
stacks_findstack(uintptr_t addr, findstack_info_t *fsip, uint_t print_warnings)
{
        mdb_findstack_kthread_t thr;
        size_t stksz;
        uintptr_t ubase, utop;
        uintptr_t kbase, ktop;
        uintptr_t win, sp;

        fsip->fsi_failed = 0;
        fsip->fsi_pc = 0;
        fsip->fsi_sp = 0;
        fsip->fsi_depth = 0;
        fsip->fsi_overflow = 0;

        if (mdb_ctf_vread(&thr, "kthread_t", "mdb_findstack_kthread_t",
            addr, print_warnings ? 0 : MDB_CTF_VREAD_QUIET) == -1) {
                fsip->fsi_failed = FSI_FAIL_BADTHREAD;
                return (DCMD_ERR);
        }

        fsip->fsi_sobj_ops = (uintptr_t)thr.t_sobj_ops;
        fsip->fsi_tstate = thr.t_state;
        fsip->fsi_panic = !!(thr.t_flag & T_PANIC);

        if ((thr.t_schedflag & TS_LOAD) == 0) {
                if (print_warnings)
                        mdb_warn("thread %p isn't in memory\n", addr);
                fsip->fsi_failed = FSI_FAIL_NOTINMEMORY;
                return (DCMD_ERR);
        }

        if (thr.t_stk < thr.t_stkbase) {
                if (print_warnings)
                        mdb_warn(
                            "stack base or stack top corrupt for thread %p\n",
                            addr);
                fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
                return (DCMD_ERR);
        }

        kbase = (uintptr_t)thr.t_stkbase;
        ktop = (uintptr_t)thr.t_stk;
        stksz = ktop - kbase;

#ifdef __amd64
        /*
         * The stack on amd64 is intentionally misaligned, so ignore the top
         * half-frame.  See thread_stk_init().  When handling traps, the frame
         * is automatically aligned by the hardware, so we only alter ktop if
         * needed.
         */
        if ((ktop & (STACK_ALIGN - 1)) != 0)
                ktop -= STACK_ENTRY_ALIGN;
#endif

        /*
         * If the stack size is larger than a meg, assume that it's bogus.
```

```
         */
        if (stksz > TOO_BIG_FOR_A_STACK) {
                if (print_warnings)
                        mdb_warn("stack size for thread %p is too big to be "
                            "reasonable\n", addr);
                fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
                return (DCMD_ERR);
        }

        /*
         * This could be (and was) a UM_GC allocation.  Unfortunately,
         * stksz tends to be very large.  As currently implemented, dcmds
         * invoked as part of pipelines don't have their UM_GC-allocated
         * memory freed until the pipeline completes.  With stksz in the
         * neighborhood of 20k, the popular ::walk thread |::findstack
         * pipeline can easily run memory-constrained debuggers (kmdb) out
         * of memory.  This can be changed back to a gc-able allocation when
         * the debugger is changed to free UM_GC memory more promptly.
         */
        ubase = (uintptr_t)mdb_alloc(stksz, UM_SLEEP);
        utop = ubase + stksz;
        if (mdb_vread((caddr_t)ubase, stksz, kbase) != stksz) {
                mdb_free((void *)ubase, stksz);
                if (print_warnings)
                        mdb_warn("couldn't read entire stack for thread %p\n",
                            addr);
                fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
                return (DCMD_ERR);
        }

        /*
         * Try the saved %sp first, if it looks reasonable.
         */
        sp = KTOU((uintptr_t)thr.t_sp + STACK_BIAS);
        if (sp >= ubase && sp <= utop) {
                if (crawl(sp, kbase, ktop, ubase, 0, fsip) == CRAWL_FOUNDALL) {
                        fsip->fsi_sp = (uintptr_t)thr.t_sp;
#if !defined(__i386)
                        fsip->fsi_pc = (uintptr_t)thr.t_pc;
#endif
                        goto found;
                }
        }

        /*
         * Now walk through the whole stack, starting at the base,
         * trying every possible "window".
         */
        for (win = ubase;
            win + sizeof (struct rwindow) <= utop;
            win += sizeof (struct rwindow *)) {
                if (crawl(win, kbase, ktop, ubase, 1, fsip) == CRAWL_FOUNDALL) {
                        fsip->fsi_sp = UTOK(win) - STACK_BIAS;
                        goto found;
                }
        }

        /*
         * We didn't conclusively find the stack.  So we'll take another lap,
         * and print out anything that looks possible.
         */
        if (print_warnings)
                mdb_printf("Possible stack pointers for thread %p:\n", addr);
        (void) mdb_vread((caddr_t)ubase, stksz, kbase);

        for (win = ubase;
	    win + sizeof (struct rwindow) <= utop;
	    win += sizeof (struct rwindow *)) {
		uintptr_t fp = ((struct rwindow *)win)->rw_fp;
		int levels;

		if ((levels = crawl(win, kbase, ktop, ubase, 1, fsip)) > 1) {
			if (print_warnings)
				mdb_printf("  %p (%d)\n", fp, levels);
		} else if (levels == CRAWL_FOUNDALL) {
			/*
			 * If this is a live system, the stack could change
			 * between the two mdb_vread(ubase, utop, kbase)'s,
			 * and we could have a fully valid stack here.
			 */
			fsip->fsi_sp = UTOK(win) - STACK_BIAS;
			goto found;
		}
	}

	fsip->fsi_depth = 0;
	fsip->fsi_overflow = 0;
	fsip->fsi_failed = FSI_FAIL_STACKNOTFOUND;

	mdb_free((void *)ubase, stksz);
	return (DCMD_ERR);
found:
	mdb_free((void *)ubase, stksz);
	return (DCMD_OK);
}
```
______***unchanged_portion_omitted_***

****
  109418 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/kmem.c
6583 remove whole-process swapping
****

______unchanged_portion_omitted_

```
static int
whatthread_walk_thread(uintptr_t addr, const kthread_t *t, whatthread_t *w)
{
        uintptr_t current, data;

        if (t->t_stkbase == NULL)
                return (WALK_NEXT);

        /*
         * Warn about swapped out threads, but drive on anyway
         */
        if (!(t->t_schedflag & TS_LOAD)) {
                mdb_warn("thread %p's stack swapped out\n", addr);
                return (WALK_NEXT);
        }

        /*
         * Search the thread's stack for the given pointer.  Note that it would
         * be more efficient to follow ::kgrep's lead and read in page-sized
         * chunks, but this routine is already fast and simple.
         */
        for (current = (uintptr_t)t->t_stkbase; current < (uintptr_t)t->t_stk;
            current += sizeof (uintptr_t)) {
                if (mdb_vread(&data, sizeof (data), current) == -1) {
                        mdb_warn("couldn't read thread %p's stack at %p",
                            addr, current);
                        return (WALK_ERR);
                }

                if (data == w->wt_target) {
                        if (w->wt_verbose) {
                                mdb_printf("%p in thread %p's stack%s\n",
                                    current, addr, stack_active(t, current));
                        } else {
                                mdb_printf("%#lr\n", addr);
                                return (WALK_NEXT);
                        }
                }
        }

        return (WALK_NEXT);
}
```

______unchanged_portion_omitted_

```
**********************************************************
   21512 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/leaky_subr.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <mdb/mdb_param.h>
#include <mdb/mdb_modapi.h>

#include <sys/fs/ufs_inode.h>
#include <sys/kmem_impl.h>
#include <sys/vmem_impl.h>
#include <sys/modctl.h>
#include <sys/kobj.h>
#include <sys/kobj_impl.h>
#include <vm/seg_vn.h>
#include <vm/as.h>
#include <vm/seg_map.h>
#include <mdb/mdb_ctf.h>

#include "kmem.h"
#include "leaky_impl.h"

/*
 * This file defines the genunix target for leaky.c.  There are three types
 * of buffers in the kernel's heap:  TYPE_VMEM, for kmem_oversize allocations,
 * TYPE_KMEM, for kmem_cache_alloc() allocations bufctl_audit_ts, and
 * TYPE_CACHE, for kmem_cache_alloc() allocation without bufctl_audit_ts.
 *
 * See "leaky_impl.h" for the target interface definition.
 */

#define TYPE_VMEM       0               /* lkb_data is the vmem_seg's size */
#define TYPE_CACHE      1               /* lkb_cid is the bufctl's cache */
#define TYPE_KMEM       2               /* lkb_cid is the bufctl's cache */

#define LKM_CTL_BUFCTL  0       /* normal allocation, PTR is bufctl */
#define LKM_CTL_VMSEG   1       /* oversize allocation, PTR is vmem_seg_t */
#define LKM_CTL_CACHE   2       /* normal alloc, non-debug, PTR is cache */
```

```
#define LKM_CTL_MASK    3L

#define LKM_CTL(ptr, type)      (LKM_CTLPTR(ptr) | (type))
#define LKM_CTLPTR(ctl)         ((uintptr_t)(ctl) & ~(LKM_CTL_MASK))
#define LKM_CTLTYPE(ctl)        ((uintptr_t)(ctl) &  (LKM_CTL_MASK))

static int kmem_lite_count = 0; /* cache of the kernel's version */

/*ARGSUSED*/
static int
leaky_mtab(uintptr_t addr, const kmem_bufctl_audit_t *bcp, leak_mtab_t **lmp)
{
        leak_mtab_t *lm = (*lmp)++;

        lm->lkm_base = (uintptr_t)bcp->bc_addr;
        lm->lkm_bufctl = LKM_CTL(addr, LKM_CTL_BUFCTL);

        return (WALK_NEXT);
}
______unchanged_portion_omitted_

/*ARGSUSED*/
#endif /* ! codereview */
static int
leaky_thread(uintptr_t addr, const kthread_t *t, unsigned long *pagesize)
{
        uintptr_t size, base = (uintptr_t)t->t_stkbase;
        uintptr_t stk = (uintptr_t)t->t_stk;

        /*
         * If this thread isn't in memory, we can't look at its stack.  This
         * may result in false positives, so we print a warning.
         */
        if (!(t->t_schedflag & TS_LOAD)) {
                mdb_printf("findleaks: thread %p's stack swapped out; "
                    "false positives possible\n", addr);
                return (WALK_NEXT);
        }

        if (t->t_state != TS_FREE)
                leaky_grep(base, stk - base);

        /*
         * There is always gunk hanging out between t_stk and the page
         * boundary.  If this thread structure wasn't kmem allocated,
         * this will include the thread structure itself.  If the thread
         * _is_ kmem allocated, we'll be able to get to it via allthreads.
         */
        size = *pagesize - (stk & (*pagesize - 1));

        leaky_grep(stk, size);

        return (WALK_NEXT);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   26490 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/rpcsvc/rstat_proc.c
patch relling-feedback
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * rstat service:  built with rstat.x
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <signal.h>
#include <utmpx.h>
#include <nlist.h>
#include <fcntl.h>
#include <syslog.h>
#include <kstat.h>

#include <rpc/rpc.h>

#include <sys/socket.h>
#include <sys/cpuvar.h>
#include <sys/sysinfo.h>
#include <sys/systm.h>
#include <errno.h>
#include <sys/stropts.h>
#include <sys/tihdr.h>
#include <sys/sysmacros.h>

#include <net/if.h>
#include <inet/mib2.h>

#include "rstat.h"
#include "rstat_v2.h"

typedef struct {
        kstat_t sys;
```

```
        kstat_t vm;
} _cpu_stats_t;
______unchanged_portion_omitted_

void
updatestat(int ignored)
{
extern int _rpcpmstart;          /* Started by a port monitor ? */
extern int _rpcsvcdirty;         /* Still serving ? */

#ifdef DEBUG
        fprintf(stderr, "entering updatestat\n");
#endif
#ifdef EXIT_WHEN_IDLE
        if (_rpcpmstart && sincelastreq >= CLOSEDOWN && !_rpcsvcdirty) {
#ifdef DEBUG
                fprintf(stderr, "about to closedown\n");
#endif
                exit(0);
        }
        sincelastreq++;
#endif /* def EXIT_WHEN_IDLE */

        (void) alarm(0);
#ifdef DEBUG
        fprintf(stderr, "boottime: %d %d\n", stats_s3.boottime.tv_sec,
                stats_s3.boottime.tv_usec);
#endif
        while (system_stat_load() || diskinfo_load() || netinfo_load()) {
                (void) kstat_chain_update(kc);
                system_stat_init();
                init_disks();
                init_net();
        }
        stats_s4.cp_time.cp_time_len = CPU_STATES;
        if (stats_s4.cp_time.cp_time_val == NULL)
                stats_s4.cp_time.cp_time_val =
                malloc(stats_s4.cp_time.cp_time_len * sizeof (int));
        stats_s2.cp_time[RSTAT_CPU_USER] =
        stats_s3.cp_time[RSTAT_CPU_USER] =
        stats_s4.cp_time.cp_time_val[RSTAT_CPU_USER] =
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_user");
        stats_s2.cp_time[RSTAT_CPU_NICE] =
        stats_s3.cp_time[RSTAT_CPU_NICE] =
        stats_s4.cp_time.cp_time_val[RSTAT_CPU_NICE] =
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_wait");
        stats_s2.cp_time[RSTAT_CPU_SYS] =
        stats_s3.cp_time[RSTAT_CPU_SYS] =
        stats_s4.cp_time.cp_time_val[RSTAT_CPU_SYS] =
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_kernel");
        stats_s2.cp_time[RSTAT_CPU_IDLE] =
        stats_s3.cp_time[RSTAT_CPU_IDLE] =
        stats_s4.cp_time.cp_time_val[RSTAT_CPU_IDLE] =
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_idle");

#ifdef DEBUG
        fprintf(stderr, "cpu: %d %d %d %d\n",
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_user"),
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_wait"),
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_kernel"),
                CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_idle"));
        fprintf(stderr, "cp_time: %d %d %d %d\n",
                stats_s3.cp_time[RSTAT_CPU_USER],
                stats_s3.cp_time[RSTAT_CPU_NICE],
                stats_s3.cp_time[RSTAT_CPU_SYS],
                stats_s3.cp_time[RSTAT_CPU_IDLE]);
```

```
#endif

        /* current time */
        gettimeofday((struct timeval *)&stats_s3.curtime, NULL);
        stats_s4.curtime = stats_s3.curtime;

        stats_s2.v_pgpgin =
        stats_s3.v_pgpgin =
        stats_s4.v_pgpgin = CPU_STAT(&cpu_stats_all.vm, "pgpgin");
        stats_s2.v_pgpgout =
        stats_s3.v_pgpgout =
        stats_s4.v_pgpgout = CPU_STAT(&cpu_stats_all.vm, "pgpgout");
        stats_s2.v_pswpin =
        stats_s3.v_pswpin =
        stats_s4.v_pswpin = 0;
        stats_s4.v_pswpin = CPU_STAT(&cpu_stats_all.vm, "pgswapin");
        stats_s2.v_pswpout =
        stats_s3.v_pswpout =
        stats_s4.v_pswpout = 0;
        stats_s4.v_pswpout = CPU_STAT(&cpu_stats_all.vm, "pgswapout");
        stats_s3.v_intr = CPU_STAT(&cpu_stats_all.sys, "intr");
        stats_s3.v_intr -= hz*(stats_s3.curtime.tv_sec - btm.tv_sec) +
                hz*(stats_s3.curtime.tv_usec - btm.tv_usec)/1000000;
        stats_s2.v_intr =
        stats_s4.v_intr = stats_s3.v_intr;
        /* swtch not in V1 */
        stats_s2.v_swtch =
        stats_s3.v_swtch =
        stats_s4.v_swtch = CPU_STAT(&cpu_stats_all.sys, "pswitch");

#ifdef DEBUG
        fprintf(stderr,
                "pgin: %d pgout: %d swpin: %d swpout: %d intr: %d swtch: %d\n",
                stats_s3.v_pgpgin,
                stats_s3.v_pgpgout,
                stats_s3.v_pswpin,
                stats_s3.v_pswpout,
                stats_s3.v_intr,
                stats_s3.v_swtch);
#endif
        /*
         * V2 and V3 of rstat are limited to RSTAT_DK_NDRIVE drives
         */
        memcpy(stats_s3.dk_xfer, stats_s4.dk_xfer.dk_xfer_val,
                RSTAT_DK_NDRIVE * sizeof (int));
        memcpy(stats_s2.dk_xfer, stats_s4.dk_xfer.dk_xfer_val,
                RSTAT_DK_NDRIVE * sizeof (int));
#ifdef DEBUG
        fprintf(stderr, "dk_xfer: %d %d %d %d\n",
                stats_s4.dk_xfer.dk_xfer_val[0],
                stats_s4.dk_xfer.dk_xfer_val[1],
                stats_s4.dk_xfer.dk_xfer_val[2],
                stats_s4.dk_xfer.dk_xfer_val[3]);
#endif

        stats_s2.if_ipackets =
        stats_s3.if_ipackets = stats_s4.if_ipackets;
        /* no s2 opackets */
        stats_s3.if_opackets = stats_s4.if_opackets;
        stats_s2.if_ierrors =
        stats_s3.if_ierrors = stats_s4.if_ierrors;
        stats_s2.if_oerrors =
        stats_s3.if_oerrors = stats_s4.if_oerrors;
        stats_s2.if_collisions =
        stats_s3.if_collisions = stats_s4.if_collisions;
```

```
        stats_s2.avenrun[0] =
        stats_s3.avenrun[0] =
        stats_s4.avenrun[0] = avenrun_1min_knp->value.ul;
        stats_s2.avenrun[1] =
        stats_s3.avenrun[1] =
        stats_s4.avenrun[1] = avenrun_5min_knp->value.ul;
        stats_s2.avenrun[2] =
        stats_s3.avenrun[2] =
        stats_s4.avenrun[2] = avenrun_15min_knp->value.ul;
#ifdef DEBUG
        fprintf(stderr, "avenrun: %d %d %d\n", stats_s3.avenrun[0],
                stats_s3.avenrun[1], stats_s3.avenrun[2]);
#endif
        signal(SIGALRM, updatestat);
        alarm(1);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   13563 Thu Feb 18 08:40:38 2016
new/usr/src/cmd/stat/vmstat/vmstat.c
patch relling-feedback
**********************************************************
______unchanged_portion_omitted_

static void
dovmstats(struct snapshot *old, struct snapshot *new)
{
        kstat_t *oldsys = NULL;
        kstat_t *newsys = &new->s_sys.ss_agg_sys;
        kstat_t *oldvm = NULL;
        kstat_t *newvm = &new->s_sys.ss_agg_vm;
        double percent_factor;
        ulong_t sys_updates, vm_updates;
        int count;

        adj = 0;

        if (old) {
                oldsys = &old->s_sys.ss_agg_sys;
                oldvm = &old->s_sys.ss_agg_vm;
        }

        etime = cpu_ticks_delta(oldsys, newsys);

        percent_factor = 100.0 / denom(etime);
        /*
         * If any time has passed, convert etime to seconds per CPU
         */
        etime = etime >= 1.0 ? (etime / nr_active_cpus(new)) / hz : 1.0;
        sys_updates = denom(DELTA(s_sys.ss_sysinfo.updates));
        vm_updates = denom(DELTA(s_sys.ss_vminfo.updates));

        if (timestamp_fmt != NODATE) {
                print_timestamp(timestamp_fmt);
                lines--;
        }

        if (--lines <= 0)
                printhdr(0);

        adj = 0;

        if (pflag) {
                adjprintf(" %*u", 6,
                    pgtok((int)(DELTA(s_sys.ss_vminfo.swap_avail)
                    / vm_updates)));
                adjprintf(" %*u", 5,
                    pgtok((int)(DELTA(s_sys.ss_vminfo.freemem) / vm_updates)));
                adjprintf(" %*.0f", 3, kstat_delta(oldvm, newvm, "pgrec")
                    / etime);
                adjprintf(" %*.0f", 3, (kstat_delta(oldvm, newvm, "hat_fault") +
                    kstat_delta(oldvm, newvm, "as_fault")) / etime);
                adjprintf(" %*.0f", 3, pgtok(kstat_delta(oldvm, newvm, "dfree"))
                    / etime);
                adjprintf(" %*ld", 3, pgtok(new->s_sys.ss_deficit));
                adjprintf(" %*.0f", 3, kstat_delta(oldvm, newvm, "scan")
                    / etime);
                adjprintf(" %*.0f", 4,
                    pgtok(kstat_delta(oldvm, newvm, "execpgin")) / etime);
                adjprintf(" %*.0f", 4,
                    pgtok(kstat_delta(oldvm, newvm, "execpgout")) / etime);
                adjprintf(" %*.0f", 4,
                    pgtok(kstat_delta(oldvm, newvm, "execfree")) / etime);
```

```
                        adjprintf(" %*.0f", 4,
                            pgtok(kstat_delta(oldvm, newvm, "anonpgin")) / etime);
                        adjprintf(" %*.0f", 4,
                            pgtok(kstat_delta(oldvm, newvm, "anonpgout")) / etime);
                        adjprintf(" %*.0f", 4,
                            pgtok(kstat_delta(oldvm, newvm, "anonfree")) / etime);
                        adjprintf(" %*.0f", 4,
                            pgtok(kstat_delta(oldvm, newvm, "fspgin")) / etime);
                        adjprintf(" %*.0f", 4,
                            pgtok(kstat_delta(oldvm, newvm, "fspgout")) / etime);
                        adjprintf(" %*.0f\n", 4,
                            pgtok(kstat_delta(oldvm, newvm, "fsfree")) / etime);
                        (void) fflush(stdout);
                        return;
                }

                adjprintf(" %*lu", 1, DELTA(s_sys.ss_sysinfo.runque) / sys_updates);
                adjprintf(" %*lu", 1, DELTA(s_sys.ss_sysinfo.waiting) / sys_updates);
                adjprintf(" %*lu", 1, DELTA(s_sys.ss_sysinfo.swpque) / sys_updates);
                adjprintf(" %*u", 6, pgtok((int)(DELTA(s_sys.ss_vminfo.swap_avail)
                    / vm_updates)));
                adjprintf(" %*u", 5, pgtok((int)(DELTA(s_sys.ss_vminfo.freemem)
                    / vm_updates)));
                adjprintf(" %*.0f", 3, swflag? 0 :
                adjprintf(" %*.0f", 3, swflag?
                    kstat_delta(oldvm, newvm, "swapin") / etime :
                    kstat_delta(oldvm, newvm, "pgrec") / etime);
                adjprintf(" %*.0f", 3, swflag? 0 :
                adjprintf(" %*.0f", 3, swflag?
                    kstat_delta(oldvm, newvm, "swapout") / etime :
                    (kstat_delta(oldvm, newvm, "hat_fault")
                    + kstat_delta(oldvm, newvm, "as_fault"))
                    / etime);
                adjprintf(" %*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "pgpgin"))
                    / etime);
                adjprintf(" %*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "pgpgout"))
                    / etime);
                adjprintf(" %*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "dfree"))
                    / etime);
                adjprintf(" %*ld", 2, pgtok(new->s_sys.ss_deficit));
                adjprintf(" %*.0f", 2, kstat_delta(oldvm, newvm, "scan") / etime);

                (void) snapshot_walk(SNAP_IODEVS, old, new, show_disk, NULL);

                count = df.if_max_iodevs - new->s_nr_iodevs;
                while (count-- > 0)
                        adjprintf(" %*d", 2, 0);

                adjprintf(" %*.0f", 4, kstat_delta(oldsys, newsys, "intr") / etime);
                adjprintf(" %*.0f", 4, kstat_delta(oldsys, newsys, "syscall") / etime);
                adjprintf(" %*.0f", 4, kstat_delta(oldsys, newsys, "pswitch") / etime);
                adjprintf(" %*.0f", 2,
                    kstat_delta(oldsys, newsys, "cpu_ticks_user") * percent_factor);
                adjprintf(" %*.0f", 2, kstat_delta(oldsys, newsys, "cpu_ticks_kernel")
                    * percent_factor);
                adjprintf(" %*.0f\n", 2, (kstat_delta(oldsys, newsys, "cpu_ticks_idle")
                    + kstat_delta(oldsys, newsys, "cpu_ticks_wait"))
                    * percent_factor);
                (void) fflush(stdout);
}
______unchanged_portion_omitted_

static void
zero_out(char const *pretty)
{
        (void) printf("%9llu %s\n", 0, pretty);
```

```
}

static void
#endif /* ! codereview */
dosum(struct sys_snapshot *ss)
{
        uint64_t total_faults;
        kstat_named_t *ksn;
        long double nchtotal;
        uint64_t nchhits;

        zero_out("swap ins");
        zero_out("swap outs");
        zero_out("pages swapped in");
        zero_out("pages swapped out");
        sum_out("swap ins", &ss->ss_agg_vm, "swapin");
        sum_out("swap outs", &ss->ss_agg_vm, "swapout");
        sum_out("pages swapped in", &ss->ss_agg_vm, "pgswapin");
        sum_out("pages swapped out", &ss->ss_agg_vm, "pgswapout");

        ksn = kstat_data_lookup(&ss->ss_agg_vm, "hat_fault");
        if (ksn == NULL) {
                fail(0, "kstat_data_lookup('%s', 'hat_fault') failed",
                    ss->ss_agg_vm.ks_name);
        }
        total_faults = ksn->value.ui64;
        ksn = kstat_data_lookup(&ss->ss_agg_vm, "as_fault");
        if (ksn == NULL) {
                fail(0, "kstat_data_lookup('%s', 'as_fault') failed",
                    ss->ss_agg_vm.ks_name);
        }
        total_faults += ksn->value.ui64;

        (void) printf("%9llu total address trans. faults taken\n",
            total_faults);

        sum_out("page ins", &ss->ss_agg_vm, "pgin");
        sum_out("page outs", &ss->ss_agg_vm, "pgout");
        sum_out("pages paged in", &ss->ss_agg_vm, "pgpgin");
        sum_out("pages paged out", &ss->ss_agg_vm, "pgpgout");
        sum_out("total reclaims", &ss->ss_agg_vm, "pgrec");
        sum_out("reclaims from free list", &ss->ss_agg_vm, "pgfrec");
        sum_out("micro (hat) faults", &ss->ss_agg_vm, "hat_fault");
        sum_out("minor (as) faults", &ss->ss_agg_vm, "as_fault");
        sum_out("major faults", &ss->ss_agg_vm, "maj_fault");
        sum_out("copy-on-write faults", &ss->ss_agg_vm, "cow_fault");
        sum_out("zero fill page faults", &ss->ss_agg_vm, "zfod");
        sum_out("pages examined by the clock daemon", &ss->ss_agg_vm, "scan");
        sum_out("revolutions of the clock hand", &ss->ss_agg_vm, "rev");
        sum_out("pages freed by the clock daemon", &ss->ss_agg_vm, "dfree");
        sum_out("forks", &ss->ss_agg_sys, "sysfork");
        sum_out("vforks", &ss->ss_agg_sys, "sysvfork");
        sum_out("execs", &ss->ss_agg_sys, "sysexec");
        sum_out("cpu context switches", &ss->ss_agg_sys, "pswitch");
        sum_out("device interrupts", &ss->ss_agg_sys, "intr");
        sum_out("traps", &ss->ss_agg_sys, "trap");
        sum_out("system calls", &ss->ss_agg_sys, "syscall");

        nchtotal = (long double) ss->ss_nc.ncs_hits.value.ui64 +
            (long double) ss->ss_nc.ncs_misses.value.ui64;
        nchhits = ss->ss_nc.ncs_hits.value.ui64;
        (void) printf("%9.0Lf total name lookups (cache hits %.0Lf%%)\n",
            nchtotal, nchhits / denom(nchtotal) * 100);

        sum_out("user   cpu", &ss->ss_agg_sys, "cpu_ticks_user");
        sum_out("system cpu", &ss->ss_agg_sys, "cpu_ticks_kernel");
```

```
        sum_out("idle   cpu", &ss->ss_agg_sys, "cpu_ticks_idle");
        sum_out("wait   cpu", &ss->ss_agg_sys, "cpu_ticks_wait");
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   66813 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/disp.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_
static void     disp_dq_alloc(struct disp_queue_info *dptr, int numpris,
    disp_t *dp);
static void     disp_dq_assign(struct disp_queue_info *dptr, int numpris);
static void     disp_dq_free(struct disp_queue_info *dptr);

/* platform-specific routine to call when processor is idle */
static void     generic_idle_cpu();
void            (*idle_cpu)() = generic_idle_cpu;

/* routines invoked when a CPU enters/exits the idle loop */
static void     idle_enter();
static void     idle_exit();

/* platform-specific routine to call when thread is enqueued */
static void     generic_enq_thread(cpu_t *, int);
void            (*disp_enq_thread)(cpu_t *, int) = generic_enq_thread;

pri_t   kpreemptpri;            /* priority where kernel preemption applies */
pri_t   upreemptpri = 0;        /* priority where normal preemption applies */
pri_t   intr_pri;               /* interrupt thread priority base level */

#define KPQPRI  -1              /* pri where cpu affinity is dropped for kpq */
pri_t   kpqpri = KPQPRI;        /* can be set in /etc/system */
disp_t  cpu0_disp;              /* boot CPU's dispatch queue */
disp_lock_t     swapped_lock;   /* lock swapped threads and swap queue */
int     nswapped;               /* total number of swapped threads */
void    disp_swapped_enq(kthread_t *tp);
static void     disp_swapped_setrun(kthread_t *tp);
static void     cpu_resched(cpu_t *cp, pri_t tpri);

/*
 * If this is set, only interrupt threads will cause kernel preemptions.
 * This is done by changing the value of kpreemptpri.  kpreemptpri
 * will either be the max sysclass pri + 1 or the min interrupt pri.
 */
int     only_intr_kpreempt;

extern void set_idle_cpu(int cpun);
extern void unset_idle_cpu(int cpun);
static void setkpdq(kthread_t *tp, int borf);
#define SETKP_BACK      0
#define SETKP_FRONT     1
/*
 * Parameter that determines how recently a thread must have run
 * on the CPU to be considered loosely-bound to that CPU to reduce
 * cold cache effects.  The interval is in hertz.
 */
#define RECHOOSE_INTERVAL 3
int     rechoose_interval = RECHOOSE_INTERVAL;

/*
 * Parameter that determines how long (in nanoseconds) a thread must
 * be sitting on a run queue before it can be stolen by another CPU
 * to reduce migrations.  The interval is in nanoseconds.
 *
 * The nosteal_nsec should be set by platform code cmp_set_nosteal_interval()
 * to an appropriate value.  nosteal_nsec is set to NOSTEAL_UNINITIALIZED
 * here indicating it is uninitiallized.
 * Setting nosteal_nsec to 0 effectively disables the nosteal 'protection'.
 *
```

```
 */
#define NOSTEAL_UNINITIALIZED   (-1)
hrtime_t nosteal_nsec = NOSTEAL_UNINITIALIZED;
extern void cmp_set_nosteal_interval(void);

id_t    defaultcid;     /* system "default" class; see dispadmin(1M) */

disp_lock_t     transition_lock;        /* lock on transitioning threads */
disp_lock_t     stop_lock;              /* lock on stopped threads */

static void     cpu_dispqalloc(int numpris);

/*
 * This gets returned by disp_getwork/disp_getbest if we couldn't steal
 * a thread because it was sitting on its run queue for a very short
 * period of time.
 */
#define T_DONTSTEAL     (kthread_t *)(-1) /* returned by disp_getwork/getbest */

static kthread_t        *disp_getwork(cpu_t *to);
static kthread_t        *disp_getbest(disp_t *from);
static kthread_t        *disp_ratify(kthread_t *tp, disp_t *kpq);

void    swtch_to(kthread_t *);

/*
 * dispatcher and scheduler initialization
 */

/*
 * disp_setup - Common code to calculate and allocate dispatcher
 *              variables and structures based on the maximum priority.
 */
static void
disp_setup(pri_t maxglobpri, pri_t oldnglobpris)
{
        pri_t   newnglobpris;

        ASSERT(MUTEX_HELD(&cpu_lock));

        newnglobpris = maxglobpri + 1 + LOCK_LEVEL;

        if (newnglobpris > oldnglobpris) {
                /*
                 * Allocate new kp queues for each CPU partition.
                 */
                cpupart_kpqalloc(newnglobpris);

                /*
                 * Allocate new dispatch queues for each CPU.
                 */
                cpu_dispqalloc(newnglobpris);

                /*
                 * compute new interrupt thread base priority
                 */
                intr_pri = maxglobpri;
                if (only_intr_kpreempt) {
                        kpreemptpri = intr_pri + 1;
                        if (kpqpri == KPQPRI)
                                kpqpri = kpreemptpri;
                }
                v.v_nglobpris = newnglobpris;
        }
}
______unchanged_portion_omitted_
```

```c
extern kthread_t *thread_unpin();

/*
 * disp() - find the highest priority thread for this processor to run, and
 * set it in TS_ONPROC state so that resume() can be called to run it.
 */
static kthread_t *
disp()
{
        cpu_t           *cpup;
        disp_t          *dp;
        kthread_t       *tp;
        dispq_t         *dq;
        int             maxrunword;
        pri_t           pri;
        disp_t          *kpq;

        TRACE_0(TR_FAC_DISP, TR_DISP_START, "disp_start");

        cpup = CPU;
        /*
         * Find the highest priority loaded, runnable thread.
         */
        dp = cpup->cpu_disp;

reschedule:
        /*
         * If there is more important work on the global queue with a better
         * priority than the maximum on this CPU, take it now.
         */
        kpq = &cpup->cpu_part->cp_kp_queue;
        while ((pri = kpq->disp_maxrunpri) >= 0 &&
            pri >= dp->disp_maxrunpri &&
            (cpup->cpu_flags & CPU_OFFLINE) == 0 &&
            (tp = disp_getbest(kpq)) != NULL) {
                if (disp_ratify(tp, kpq) != NULL) {
                        TRACE_1(TR_FAC_DISP, TR_DISP_END,
                            "disp_end:tid %p", tp);
                        return (tp);
                }
        }

        disp_lock_enter(&dp->disp_lock);
        pri = dp->disp_maxrunpri;

        /*
         * If there is nothing to run, look at what's runnable on other queues.
         * Choose the idle thread if the CPU is quiesced.
         * Note that CPUs that have the CPU_OFFLINE flag set can still run
         * interrupt threads, which will be the only threads on the CPU's own
         * queue, but cannot run threads from other queues.
         */
        if (pri == -1) {
                if (!(cpup->cpu_flags & CPU_OFFLINE)) {
                        disp_lock_exit(&dp->disp_lock);
                        if ((tp = disp_getwork(cpup)) == NULL ||
                            tp == T_DONTSTEAL) {
                                tp = cpup->cpu_idle_thread;
                                (void) splhigh();
                                THREAD_ONPROC(tp, cpup);
                                cpup->cpu_dispthread = tp;
                                cpup->cpu_dispatch_pri = -1;
                                cpup->cpu_runrun = cpup->cpu_kprunrun = 0;
                                cpup->cpu_chosen_level = -1;
                        }
```

```c
                } else {
                        disp_lock_exit_high(&dp->disp_lock);
                        tp = cpup->cpu_idle_thread;
                        THREAD_ONPROC(tp, cpup);
                        cpup->cpu_dispthread = tp;
                        cpup->cpu_dispatch_pri = -1;
                        cpup->cpu_runrun = cpup->cpu_kprunrun = 0;
                        cpup->cpu_chosen_level = -1;
                }
                TRACE_1(TR_FAC_DISP, TR_DISP_END,
                    "disp_end:tid %p", tp);
                return (tp);
        }

        dq = &dp->disp_q[pri];
        tp = dq->dq_first;

        ASSERT(tp != NULL);
        ASSERT(tp->t_schedflag & TS_LOAD);      /* thread must be swapped in */

        DTRACE_SCHED2(dequeue, kthread_t *, tp, disp_t *, dp);

        /*
         * Found it so remove it from queue.
         */
        dp->disp_nrunnable--;
        dq->dq_sruncnt--;
        if ((dq->dq_first = tp->t_link) == NULL) {
                ulong_t *dqactmap = dp->disp_qactmap;

                ASSERT(dq->dq_sruncnt == 0);
                dq->dq_last = NULL;

                /*
                 * The queue is empty, so the corresponding bit needs to be
                 * turned off in dqactmap.   If nrunnable != 0 just took the
                 * last runnable thread off the
                 * highest queue, so recompute disp_maxrunpri.
                 */
                maxrunword = pri >> BT_ULSHIFT;
                dqactmap[maxrunword] &= ~BT_BIW(pri);

                if (dp->disp_nrunnable == 0) {
                        dp->disp_max_unbound_pri = -1;
                        dp->disp_maxrunpri = -1;
                } else {
                        int ipri;

                        ipri = bt_gethighbit(dqactmap, maxrunword);
                        dp->disp_maxrunpri = ipri;
                        if (ipri < dp->disp_max_unbound_pri)
                                dp->disp_max_unbound_pri = ipri;
                }
        } else {
                tp->t_link = NULL;
        }

        /*
         * Set TS_DONT_SWAP flag to prevent another processor from swapping
         * out this thread before we have a chance to run it.
         * While running, it is protected against swapping by t_lock.
         */
        tp->t_schedflag |= TS_DONT_SWAP;
        cpup->cpu_dispthread = tp;              /* protected by spl only */
        cpup->cpu_dispatch_pri = pri;
        ASSERT(pri == DISP_PRIO(tp));
        thread_onproc(tp, cpup);                /* set t_state to TS_ONPROC */
        disp_lock_exit_high(&dp->disp_lock);    /* drop run queue lock */

        ASSERT(tp != NULL);
        TRACE_1(TR_FAC_DISP, TR_DISP_END,
            "disp_end:tid %p", tp);

        if (disp_ratify(tp, kpq) == NULL)
                goto reschedule;

        return (tp);
}
```

______unchanged_portion_omitted_

```
/*
 * setbackdq() keeps runqs balanced such that the difference in length
 * between the chosen runq and the next one is no more than RUNQ_MAX_DIFF.
 * For threads with priorities below RUNQ_MATCH_PRI levels, the runq's lengths
 * must match.  When per-thread TS_RUNQMATCH flag is set, setbackdq() will
 * try to keep runqs perfectly balanced regardless of the thread priority.
 */
#define RUNQ_MATCH_PRI  16      /* pri below which queue lengths must match */
#define RUNQ_MAX_DIFF   2       /* maximum runq length difference */
#define RUNQ_LEN(cp, pri)       ((cp)->cpu_disp->disp_q[pri].dq_sruncnt)

/*
 * Macro that evaluates to true if it is likely that the thread has cache
 * warmth. This is based on the amount of time that has elapsed since the
 * thread last ran. If that amount of time is less than "rechoose_interval"
 * ticks, then we decide that the thread has enough cache warmth to warrant
 * some affinity for t->t_cpu.
 */
#define THREAD_HAS_CACHE_WARMTH(thread) \
        ((thread == curthread) ||       \
        ((ddi_get_lbolt() - thread->t_disp_time) <= rechoose_interval))
/*
 * Put the specified thread on the back of the dispatcher
 * queue corresponding to its current priority.
 *
 * Called with the thread in transition, onproc or stopped state
 * and locked (transition implies locked) and at high spl.
 * Returns with the thread in TS_RUN state and still locked.
 */
void
setbackdq(kthread_t *tp)
{
        dispq_t *dq;
        disp_t          *dp;
        cpu_t           *cp;
        pri_t           tpri;
        int             bound;
        boolean_t       self;

        ASSERT(THREAD_LOCK_HELD(tp));
        ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
        ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

        /*
         * If thread is "swapped" or on the swap queue don't
         * queue it, but wake sched.
         */
        if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
                disp_swapped_setrun(tp);
                return;
        }
```

```
        self = (tp == curthread);

        if (tp->t_bound_cpu || tp->t_weakbound_cpu)
                bound = 1;
        else
                bound = 0;

        tpri = DISP_PRIO(tp);
        if (ncpus == 1)
                cp = tp->t_cpu;
        else if (!bound) {
                if (tpri >= kpqpri) {
                        setkpdq(tp, SETKP_BACK);
                        return;
                }

                /*
                 * We'll generally let this thread continue to run where
                 * it last ran...but will consider migration if:
                 * - We thread probably doesn't have much cache warmth.
                 * - The CPU where it last ran is the target of an offline
                 *   request.
                 * - The thread last ran outside it's home lgroup.
                 */
                if ((!THREAD_HAS_CACHE_WARMTH(tp)) ||
                    (tp->t_cpu == cpu_inmotion)) {
                        cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri, NULL);
                } else if (!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, tp->t_cpu)) {
                        cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
                            self ? tp->t_cpu : NULL);
                } else {
                        cp = tp->t_cpu;
                }

                if (tp->t_cpupart == cp->cpu_part) {
                        int     qlen;

                        /*
                         * Perform any CMT load balancing
                         */
                        cp = cmt_balance(tp, cp);

                        /*
                         * Balance across the run queues
                         */
                        qlen = RUNQ_LEN(cp, tpri);
                        if (tpri >= RUNQ_MATCH_PRI &&
                            !(tp->t_schedflag & TS_RUNQMATCH))
                                qlen -= RUNQ_MAX_DIFF;
                        if (qlen > 0) {
                                cpu_t *newcp;

                                if (tp->t_lpl->lpl_lgrpid == LGRP_ROOTID) {
                                        newcp = cp->cpu_next_part;
                                } else if ((newcp = cp->cpu_next_lpl) == cp) {
                                        newcp = cp->cpu_next_part;
                                }

                                if (RUNQ_LEN(newcp, tpri) < qlen) {
                                        DTRACE_PROBE3(runq__balance,
                                            kthread_t *, tp,
                                            cpu_t *, cp, cpu_t *, newcp);
                                        cp = newcp;
                                }
                        }
                } else {
                        /*
                         * Migrate to a cpu in the new partition.
                         */
                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
                            tp->t_lpl, tp->t_pri, NULL);
                }
                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
        } else {
                /*
                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
                 * a short time until weak binding that existed when the
                 * strong binding was established has dropped) so we must
                 * favour weak binding over strong.
                 */
                cp = tp->t_weakbound_cpu ?
                    tp->t_weakbound_cpu : tp->t_bound_cpu;
        }
        /*
         * A thread that is ONPROC may be temporarily placed on the run queue
         * but then chosen to run again by disp.  If the thread we're placing on
         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
         * replacement process is actually scheduled in swtch().  In this
         * situation, curthread is the only thread that could be in the ONPROC
         * state.
         */
        if ((!self) && (tp->t_waitrq == 0)) {
                hrtime_t curtime;

                curtime = gethrtime_unscaled();
                (void) cpu_update_pct(tp, curtime);
                tp->t_waitrq = curtime;
        } else {
                (void) cpu_update_pct(tp, gethrtime_unscaled());
        }

        dp = cp->cpu_disp;
        disp_lock_enter_high(&dp->disp_lock);

        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 0);
        TRACE_3(TR_FAC_DISP, TR_BACKQ, "setbackdq:pri %d cpu %p tid %p",
            tpri, cp, tp);

#ifndef NPROBE
        /* Kernel probe */
        if (tnf_tracing_active)
                tnf_thread_queue(tp, cp, tpri);
#endif /* NPROBE */

        ASSERT(tpri >= 0 && tpri < dp->disp_npri);

        THREAD_RUN(tp, &dp->disp_lock);         /* set t_state to TS_RUN */
        tp->t_disp_queue = dp;
        tp->t_link = NULL;

        dq = &dp->disp_q[tpri];
        dp->disp_nrunnable++;
        if (!bound)
                dp->disp_steal = 0;
        membar_enter();

        if (dq->dq_sruncnt++ != 0) {
                ASSERT(dq->dq_first != NULL);
                dq->dq_last->t_link = tp;
                dq->dq_last = tp;
        } else {
                ASSERT(dq->dq_first == NULL);
```

```
                ASSERT(dq->dq_last == NULL);
                dq->dq_first = dq->dq_last = tp;
                BT_SET(dp->disp_qactmap, tpri);
                if (tpri > dp->disp_maxrunpri) {
                        dp->disp_maxrunpri = tpri;
                        membar_enter();
                        cpu_resched(cp, tpri);
                }
        }

        if (!bound && tpri > dp->disp_max_unbound_pri) {
                if (self && dp->disp_max_unbound_pri == -1 && cp == CPU) {
                        /*
                         * If there are no other unbound threads on the
                         * run queue, don't allow other CPUs to steal
                         * this thread while we are in the middle of a
                         * context switch. We may just switch to it
                         * again right away. CPU_DISP_DONTSTEAL is cleared
                         * in swtch and swtch_to.
                         */
                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
                }
                dp->disp_max_unbound_pri = tpri;
        }
        (*disp_enq_thread)(cp, bound);
}

/*
 * Put the specified thread on the front of the dispatcher
 * queue corresponding to its current priority.
 *
 * Called with the thread in transition, onproc or stopped state
 * and locked (transition implies locked) and at high spl.
 * Returns with the thread in TS_RUN state and still locked.
 */
void
setfrontdq(kthread_t *tp)
{
        disp_t          *dp;
        dispq_t         *dq;
        cpu_t           *cp;
        pri_t           tpri;
        int             bound;

        ASSERT(THREAD_LOCK_HELD(tp));
        ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
        ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

        /*
         * If thread is "swapped" or on the swap queue don't
         * queue it, but wake sched.
         */
        if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
                disp_swapped_setrun(tp);
                return;
        }

        if (tp->t_bound_cpu || tp->t_weakbound_cpu)
                bound = 1;
        else
                bound = 0;

        tpri = DISP_PRIO(tp);
        if (ncpus == 1)
                cp = tp->t_cpu;
        else if (!bound) {
                if (tpri >= kpqpri) {
                        setkpdq(tp, SETKP_FRONT);
                        return;
                }
                cp = tp->t_cpu;
                if (tp->t_cpupart == cp->cpu_part) {
                        /*
                         * We'll generally let this thread continue to run
                         * where it last ran, but will consider migration if:
                         * - The thread last ran outside it's home lgroup.
                         * - The CPU where it last ran is the target of an
                         *   offline request (a thread_nomigrate() on the in
                         *   motion CPU relies on this when forcing a preempt).
                         * - The thread isn't the highest priority thread where
                         *   it last ran, and it is considered not likely to
                         *   have significant cache warmth.
                         */
                        if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)) ||
                            (cp == cpu_inmotion)) {
                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
                                    (tp == curthread) ? cp : NULL);
                        } else if ((tpri < cp->cpu_disp->disp_maxrunpri) &&
                            (!THREAD_HAS_CACHE_WARMTH(tp))) {
                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
                                    NULL);
                        }
                } else {
                        /*
                         * Migrate to a cpu in the new partition.
                         */
                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
                            tp->t_lpl, tp->t_pri, NULL);
                }
                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
        } else {
                /*
                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
                 * a short time until weak binding that existed when the
                 * strong binding was established has dropped) so we must
                 * favour weak binding over strong.
                 */
                cp = tp->t_weakbound_cpu ?
                    tp->t_weakbound_cpu : tp->t_bound_cpu;
        }

        /*
         * A thread that is ONPROC may be temporarily placed on the run queue
         * but then chosen to run again by disp.  If the thread we're placing on
         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
         * replacement process is actually scheduled in swtch().  In this
         * situation, curthread is the only thread that could be in the ONPROC
         * state.
         */
        if ((tp != curthread) && (tp->t_waitrq == 0)) {
                hrtime_t curtime;

                curtime = gethrtime_unscaled();
                (void) cpu_update_pct(tp, curtime);
                tp->t_waitrq = curtime;
        } else {
                (void) cpu_update_pct(tp, gethrtime_unscaled());
        }

        dp = cp->cpu_disp;
        disp_lock_enter_high(&dp->disp_lock);
```

```
        TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri, tp);
        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 1);

#ifndef NPROBE
        /* Kernel probe */
        if (tnf_tracing_active)
                tnf_thread_queue(tp, cp, tpri);
#endif /* NPROBE */

        ASSERT(tpri >= 0 && tpri < dp->disp_npri);

        THREAD_RUN(tp, &dp->disp_lock);         /* set TS_RUN state and lock */
        tp->t_disp_queue = dp;

        dq = &dp->disp_q[tpri];
        dp->disp_nrunnable++;
        if (!bound)
                dp->disp_steal = 0;
        membar_enter();

        if (dq->dq_sruncnt++ != 0) {
                ASSERT(dq->dq_last != NULL);
                tp->t_link = dq->dq_first;
                dq->dq_first = tp;
        } else {
                ASSERT(dq->dq_last == NULL);
                ASSERT(dq->dq_first == NULL);
                tp->t_link = NULL;
                dq->dq_first = dq->dq_last = tp;
                BT_SET(dp->disp_qactmap, tpri);
                if (tpri > dp->disp_maxrunpri) {
                        dp->disp_maxrunpri = tpri;
                        membar_enter();
                        cpu_resched(cp, tpri);
                }
        }

        if (!bound && tpri > dp->disp_max_unbound_pri) {
                if (tp == curthread && dp->disp_max_unbound_pri == -1 &&
                    cp == CPU) {
                        /*
                         * If there are no other unbound threads on the
                         * run queue, don't allow other CPUs to steal
                         * this thread while we are in the middle of a
                         * context switch. We may just switch to it
                         * again right away. CPU_DISP_DONTSTEAL is cleared
                         * in swtch and swtch_to.
                         */
                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
                }
                dp->disp_max_unbound_pri = tpri;
        }
        (*disp_enq_thread)(cp, bound);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Remove a thread from the dispatcher queue if it is on it.
 * It is not an error if it is not found but we return whether
 * or not it was found in case the caller wants to check.
 */
int
dispdeq(kthread_t *tp)
{
        disp_t          *dp;
        dispq_t         *dq;
	kthread_t	*rp;
	kthread_t	*trp;
	kthread_t	**ptp;
	int		tpri;

	ASSERT(THREAD_LOCK_HELD(tp));

	if (tp->t_state != TS_RUN)
		return (0);

	/*
	 * The thread is "swapped" or is on the swap queue and
	 * hence no longer on the run queue, so return true.
	 */
	if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD)
		return (1);

	tpri = DISP_PRIO(tp);
	dp = tp->t_disp_queue;
	ASSERT(tpri < dp->disp_npri);
	dq = &dp->disp_q[tpri];
	ptp = &dq->dq_first;
	rp = *ptp;
	trp = NULL;

	ASSERT(dq->dq_last == NULL || dq->dq_last->t_link == NULL);

	/*
	 * Search for thread in queue.
	 * Double links would simplify this at the expense of disp/setrun.
	 */
	while (rp != tp && rp != NULL) {
		trp = rp;
		ptp = &trp->t_link;
		rp = trp->t_link;
	}

	if (rp == NULL) {
		panic("dispdeq: thread not on queue");
	}

	DTRACE_SCHED2(dequeue, kthread_t *, tp, disp_t *, dp);

	/*
	 * Found it so remove it from queue.
	 */
	if ((*ptp = rp->t_link) == NULL)
		dq->dq_last = trp;

	dp->disp_nrunnable--;
	if (--dq->dq_sruncnt == 0) {
		dp->disp_qactmap[tpri >> BT_ULSHIFT] &= ~BT_BIW(tpri);
		if (dp->disp_nrunnable == 0) {
			dp->disp_max_unbound_pri = -1;
			dp->disp_maxrunpri = -1;
		} else if (tpri == dp->disp_maxrunpri) {
			int ipri;

			ipri = bt_gethighbit(dp->disp_qactmap,
			    dp->disp_maxrunpri >> BT_ULSHIFT);
			if (ipri < dp->disp_max_unbound_pri)
				dp->disp_max_unbound_pri = ipri;
			dp->disp_maxrunpri = ipri;
		}
	}
	tp->t_link = NULL;
```

```
	THREAD_TRANSITION(tp);		/* put in intermediate state */
	return (1);
}


/*
 * dq_sruninc and dq_srundec are public functions for
 * incrementing/decrementing the sruncnts when a thread on
 * a dispatcher queue is made schedulable/unschedulable by
 * resetting the TS_LOAD flag.
 *
 * The caller MUST have the thread lock and therefore the dispatcher
 * queue lock so that the operation which changes
 * the flag, the operation that checks the status of the thread to
 * determine if it's on a disp queue AND the call to this function
 * are one atomic operation with respect to interrupts.
 */

/*
 * Called by sched AFTER TS_LOAD flag is set on a swapped, runnable thread.
 */
void
dq_sruninc(kthread_t *t)
{
	ASSERT(t->t_state == TS_RUN);
	ASSERT(t->t_schedflag & TS_LOAD);

	THREAD_TRANSITION(t);
	setfrontdq(t);
}

/*
 * See comment on calling conventions above.
 * Called by sched BEFORE TS_LOAD flag is cleared on a runnable thread.
 */
void
dq_srundec(kthread_t *t)
{
	ASSERT(t->t_schedflag & TS_LOAD);

	(void) dispdeq(t);
	disp_swapped_enq(t);
}

/*
 * Change the dispatcher lock of thread to the "swapped_lock"
 * and return with thread lock still held.
 *
 * Called with thread_lock held, in transition state, and at high spl.
 */
void
disp_swapped_enq(kthread_t *tp)
{
	ASSERT(THREAD_LOCK_HELD(tp));
	ASSERT(tp->t_schedflag & TS_LOAD);

	switch (tp->t_state) {
	case TS_RUN:
		disp_lock_enter_high(&swapped_lock);
		THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */
		break;
	case TS_ONPROC:
		disp_lock_enter_high(&swapped_lock);
		THREAD_TRANSITION(tp);
		wake_sched_sec = 1;		/* tell clock to wake sched */
		THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */
```

```
                break;
        default:
                panic("disp_swapped: tp: %p bad t_state", (void *)tp);
        }
}

/*
 * This routine is called by setbackdq/setfrontdq if the thread is
 * not loaded or loaded and on the swap queue.
 *
 * Thread state TS_SLEEP implies that a swapped thread
 * has been woken up and needs to be swapped in by the swapper.
 *
 * Thread state TS_RUN, it implies that the priority of a swapped
 * thread is being increased by scheduling class (e.g. ts_update).
 */
static void
disp_swapped_setrun(kthread_t *tp)
{
        ASSERT(THREAD_LOCK_HELD(tp));
        ASSERT((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD);

        switch (tp->t_state) {
        case TS_SLEEP:
                disp_lock_enter_high(&swapped_lock);
                /*
                 * Wakeup sched immediately (i.e., next tick) if the
                 * thread priority is above maxclsyspri.
                 */
                if (DISP_PRIO(tp) > maxclsyspri)
                        wake_sched = 1;
                else
                        wake_sched_sec = 1;
                THREAD_RUN(tp, &swapped_lock); /* set TS_RUN state and lock */
                break;
        case TS_RUN:                            /* called from ts_update */
                break;
        default:
                panic("disp_swapped_setrun: tp: %p bad t_state", (void *)tp);
        }
}

/*
 *      Make a thread give up its processor.  Find the processor on
 *      which this thread is executing, and have that processor
 *      preempt.
 *
 *      We allow System Duty Cycle (SDC) threads to be preempted even if
 *      they are running at kernel priorities.  To implement this, we always
 *      set cpu_kprunrun; this ensures preempt() will be called.  Since SDC
 *      calls cpu_surrender() very often, we only preempt if there is anyone
 *      competing with us.
 */
void
cpu_surrender(kthread_t *tp)
{
        cpu_t   *cpup;
        int     max_pri;
        int     max_run_pri;
        klwp_t  *lwp;

        ASSERT(THREAD_LOCK_HELD(tp));

        if (tp->t_state != TS_ONPROC)
                return;
        cpup = tp->t_disp_queue->disp_cpu;      /* CPU thread dispatched to */
```

```
        max_pri = cpup->cpu_disp->disp_maxrunpri; /* best pri of that CPU */
        max_run_pri = CP_MAXRUNPRI(cpup->cpu_part);
        if (max_pri < max_run_pri)
                max_pri = max_run_pri;

        if (tp->t_cid == sysdccid) {
                uint_t t_pri = DISP_PRIO(tp);
                if (t_pri > max_pri)
                        return;         /* we are not competing w/ anyone */
                cpup->cpu_runrun = cpup->cpu_kprunrun = 1;
        } else {
                cpup->cpu_runrun = 1;
                if (max_pri >= kpreemptpri && cpup->cpu_kprunrun == 0) {
                        cpup->cpu_kprunrun = 1;
                }
        }

        /*
         * Propagate cpu_runrun, and cpu_kprunrun to global visibility.
         */
        membar_enter();

        DTRACE_SCHED1(surrender, kthread_t *, tp);

        /*
         * Make the target thread take an excursion through trap()
         * to do preempt() (unless we're already in trap or post_syscall,
         * calling cpu_surrender via CL_TRAPRET).
         */
        if (tp != curthread || (lwp = tp->t_lwp) == NULL ||
            lwp->lwp_state != LWP_USER) {
                aston(tp);
                if (cpup != CPU)
                        poke_cpu(cpup->cpu_id);
        }
        TRACE_2(TR_FAC_DISP, TR_CPU_SURRENDER,
            "cpu_surrender:tid %p cpu %p", tp, cpup);
}
______unchanged_portion_omitted_

/*
 * disp_adjust_unbound_pri() - thread is becoming unbound, so we should
 *      check if the CPU to which is was previously bound should have
 *      its disp_max_unbound_pri increased.
 */
void
disp_adjust_unbound_pri(kthread_t *tp)
{
        disp_t *dp;
        pri_t tpri;

        ASSERT(THREAD_LOCK_HELD(tp));

        /*
         * Don't do anything if the thread is not bound, or
         * currently not runnable.
         * currently not runnable or swapped out.
         */
        if (tp->t_bound_cpu == NULL ||
            tp->t_state != TS_RUN)
            tp->t_state != TS_RUN ||
            tp->t_schedflag & TS_ON_SWAPQ)
                return;

        tpri = DISP_PRIO(tp);
        dp = tp->t_bound_cpu->cpu_disp;
        ASSERT(tpri >= 0 && tpri < dp->disp_npri);
        if (tpri > dp->disp_max_unbound_pri)
                dp->disp_max_unbound_pri = tpri;
}

/*
 * disp_getbest()
 *   De-queue the highest priority unbound runnable thread.
 *   Returns with the thread unlocked and onproc but at splhigh (like disp()).
 *   Returns NULL if nothing found.
 *   Returns T_DONTSTEAL if the thread was not stealable.
 *   so that the caller will try again later.
 *
 *   Passed a pointer to a dispatch queue not associated with this CPU, and
 *   its type.
 */
static kthread_t *
disp_getbest(disp_t *dp)
{
        kthread_t       *tp;
        dispq_t         *dq;
        pri_t           pri;
        cpu_t           *cp, *tcp;
        boolean_t       allbound;

        disp_lock_enter(&dp->disp_lock);

        /*
         * If there is nothing to run, or the CPU is in the middle of a
         * context switch of the only thread, return NULL.
         */
        tcp = dp->disp_cpu;
        cp = CPU;
        pri = dp->disp_max_unbound_pri;
        if (pri == -1 ||
            (tcp != NULL && (tcp->cpu_disp_flags & CPU_DISP_DONTSTEAL) &&
            tcp->cpu_disp->disp_nrunnable == 1)) {
                disp_lock_exit_nopreempt(&dp->disp_lock);
                return (NULL);
        }

        dq = &dp->disp_q[pri];


        /*
         * Assume that all threads are bound on this queue, and change it
         * later when we find out that it is not the case.
         */
        allbound = B_TRUE;
        for (tp = dq->dq_first; tp != NULL; tp = tp->t_link) {
                hrtime_t now, nosteal, rqtime;

                /*
                 * Skip over bound threads which could be here even
                 * though disp_max_unbound_pri indicated this level.
                 */
                if (tp->t_bound_cpu || tp->t_weakbound_cpu)
                        continue;

                /*
                 * We’ve got some unbound threads on this queue, so turn
                 * the allbound flag off now.
                 */
                allbound = B_FALSE;

                /*
```

```
                 * The thread is a candidate for stealing from its run queue. We
                 * don’t want to steal threads that became runnable just a
                 * moment ago. This improves CPU affinity for threads that get
                 * preempted for short periods of time and go back on the run
                 * queue.
                 *
                 * We want to let it stay on its run queue if it was only placed
                 * there recently and it was running on the same CPU before that
                 * to preserve its cache investment. For the thread to remain on
                 * its run queue, ALL of the following conditions must be
                 * satisfied:
                 *
                 * - the disp queue should not be the kernel preemption queue
                 * - delayed idle stealing should not be disabled
                 * - nosteal_nsec should be non-zero
                 * - it should run with user priority
                 * - it should be on the run queue of the CPU where it was
                 *   running before being placed on the run queue
                 * - it should be the only thread on the run queue (to prevent
                 *   extra scheduling latency for other threads)
                 * - it should sit on the run queue for less than per-chip
                 *   nosteal interval or global nosteal interval
                 * - in case of CPUs with shared cache it should sit in a run
                 *   queue of a CPU from a different chip
                 *
                 * The checks are arranged so that the ones that are faster are
                 * placed earlier.
                 */
                if (tcp == NULL ||
                    pri >= minclsyspri ||
                    tp->t_cpu != tcp)
                        break;

                /*
                 * Steal immediately if, due to CMT processor architecture
                 * migraiton between cp and tcp would incur no performance
                 * penalty.
                 */
                if (pg_cmt_can_migrate(cp, tcp))
                        break;

                nosteal = nosteal_nsec;
                if (nosteal == 0)
                        break;

                /*
                 * Calculate time spent sitting on run queue
                 */
                now = gethrtime_unscaled();
                rqtime = now - tp->t_waitrq;
                scalehrtime(&rqtime);

                /*
                 * Steal immediately if the time spent on this run queue is more
                 * than allowed nosteal delay.
                 *
                 * Negative rqtime check is needed here to avoid infinite
                 * stealing delays caused by unlikely but not impossible
                 * drifts between CPU times on different CPUs.
                 */
                if (rqtime > nosteal || rqtime < 0)
                        break;

                DTRACE_PROBE4(nosteal, kthread_t *, tp,
                    cpu_t *, tcp, cpu_t *, cp, hrtime_t, rqtime);
                scalehrtime(&now);
```

```
		/*
		 * Calculate when this thread becomes stealable
		 */
		now += (nosteal - rqtime);

		/*
		 * Calculate time when some thread becomes stealable
		 */
		if (now < dp->disp_steal)
			dp->disp_steal = now;
	}

	/*
	 * If there were no unbound threads on this queue, find the queue
	 * where they are and then return later. The value of
	 * disp_max_unbound_pri is not always accurate because it isn't
	 * reduced until another idle CPU looks for work.
	 */
	if (allbound)
		disp_fix_unbound_pri(dp, pri);

	/*
	 * If we reached the end of the queue and found no unbound threads
	 * then return NULL so that other CPUs will be considered.  If there
	 * are unbound threads but they cannot yet be stolen, then
	 * return T_DONTSTEAL and try again later.
	 */
	if (tp == NULL) {
		disp_lock_exit_nopreempt(&dp->disp_lock);
		return (allbound ? NULL : T_DONTSTEAL);
	}

	/*
	 * Found a runnable, unbound thread, so remove it from queue.
	 * dispdeq() requires that we have the thread locked, and we do,
	 * by virtue of holding the dispatch queue lock.  dispdeq() will
	 * put the thread in transition state, thereby dropping the dispq
	 * lock.
	 */

#ifdef DEBUG
	{
		int	thread_was_on_queue;

		thread_was_on_queue = dispdeq(tp);	/* drops disp_lock */
		ASSERT(thread_was_on_queue);
	}

#else /* DEBUG */
	(void) dispdeq(tp);			/* drops disp_lock */
#endif /* DEBUG */

	/*
	 * Reset the disp_queue steal time - we do not know what is the smallest
	 * value across the queue is.
	 */
	dp->disp_steal = 0;

	tp->t_schedflag |= TS_DONT_SWAP;

	/*
	 * Setup thread to run on the current CPU.
	 */
	tp->t_disp_queue = cp->cpu_disp;

	cp->cpu_dispthread = tp;		/* protected by spl only */
```

```
	cp->cpu_dispatch_pri = pri;

	/*
	 * There can be a memory synchronization race between disp_getbest()
	 * and disp_ratify() vs cpu_resched() where cpu_resched() is trying
	 * to preempt the current thread to run the enqueued thread while
	 * disp_getbest() and disp_ratify() are changing the current thread
	 * to the stolen thread. This may lead to a situation where
	 * cpu_resched() tries to preempt the wrong thread and the
	 * stolen thread continues to run on the CPU which has been tagged
	 * for preemption.
	 * Later the clock thread gets enqueued but doesn't get to run on the
	 * CPU causing the system to hang.
	 *
	 * To avoid this, grabbing and dropping the disp_lock (which does
	 * a memory barrier) is needed to synchronize the execution of
	 * cpu_resched() with disp_getbest() and disp_ratify() and
	 * synchronize the memory read and written by cpu_resched(),
	 * disp_getbest(), and disp_ratify() with each other.
	 *  (see CR#6482861 for more details).
	 */
	disp_lock_enter_high(&cp->cpu_disp->disp_lock);
	disp_lock_exit_high(&cp->cpu_disp->disp_lock);

	ASSERT(pri == DISP_PRIO(tp));

	DTRACE_PROBE3(steal, kthread_t *, tp, cpu_t *, tcp, cpu_t *, cp);

	thread_onproc(tp, cp);			/* set t_state to TS_ONPROC */

	/*
	 * Return with spl high so that swtch() won't need to raise it.
	 * The disp_lock was dropped by dispdeq().
	 */

	return (tp);
}
______unchanged_portion_omitted_
```

****
   79977 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/fss.c
6583 remove whole-process swapping
****
______***unchanged_portion_omitted_***

```
#define FSS_TICK_COST   1000    /* tick cost for threads with nice level = 0 */

/*
 * Decay rate percentages are based on n/128 rather than n/100 so  that
 * calculations can avoid having to do an integer divide by 100 (divide
 * by FSS_DECAY_BASE == 128 optimizes to an arithmetic shift).
 *
 * FSS_DECAY_MIN        =  83/128 ~= 65%
 * FSS_DECAY_MAX        = 108/128 ~= 85%
 * FSS_DECAY_USG        =  96/128 ~= 75%
 */
#define FSS_DECAY_MIN   83      /* fsspri decay pct for threads w/ nice -20 */
#define FSS_DECAY_MAX   108     /* fsspri decay pct for threads w/ nice +19 */
#define FSS_DECAY_USG   96      /* fssusage decay pct for projects */
#define FSS_DECAY_BASE  128     /* base for decay percentages above */

#define FSS_NICE_MIN    0
#define FSS_NICE_MAX    (2 * NZERO - 1)
#define FSS_NICE_RANGE  (FSS_NICE_MAX - FSS_NICE_MIN + 1)

static int      fss_nice_tick[FSS_NICE_RANGE];
static int      fss_nice_decay[FSS_NICE_RANGE];

static pri_t    fss_maxupri = FSS_MAXUPRI; /* maximum FSS user priority */
static pri_t    fss_maxumdpri; /* maximum user mode fss priority */
static pri_t    fss_maxglobpri; /* maximum global priority used by fss class */
static pri_t    fss_minglobpri; /* minimum global priority */

static fssproc_t fss_listhead[FSS_LISTS];
static kmutex_t fss_listlock[FSS_LISTS];

static fsspset_t *fsspsets;
static kmutex_t fsspsets_lock;  /* protects fsspsets */

static id_t     fss_cid;

static time_t   fss_minrun = 2; /* t_pri becomes 59 within 2 secs */
static time_t   fss_minslp = 2; /* min time on sleep queue for hardswap */
static int      fss_quantum = 11;

static void     fss_newpri(fssproc_t *, boolean_t);
static void     fss_update(void *);
static int      fss_update_list(int);
static void     fss_change_priority(kthread_t *, fssproc_t *);

static int      fss_admin(caddr_t, cred_t *);
static int      fss_getclinfo(void *);
static int      fss_parmsin(void *);
static int      fss_parmsout(void *, pc_vaparms_t *);
static int      fss_vaparmsin(void *, pc_vaparms_t *);
static int      fss_vaparmsout(void *, pc_vaparms_t *);
static int      fss_getclpri(pcpri_t *);
static int      fss_alloc(void **, int);
static void     fss_free(void *);

static int      fss_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static void     fss_exitclass(void *);
static int      fss_canexit(kthread_t *, cred_t *);
static int      fss_fork(kthread_t *, kthread_t *, void *);
```

```
static void     fss_forkret(kthread_t *, kthread_t *);
static void     fss_parmsget(kthread_t *, void *);
static int      fss_parmsset(kthread_t *, void *, id_t, cred_t *);
static void     fss_stop(kthread_t *, int, int);
static void     fss_exit(kthread_t *);
static void     fss_active(kthread_t *);
static void     fss_inactive(kthread_t *);
static pri_t    fss_swapin(kthread_t *, int);
static pri_t    fss_swapout(kthread_t *, int);
static void     fss_trapret(kthread_t *);
static void     fss_preempt(kthread_t *);
static void     fss_setrun(kthread_t *);
static void     fss_sleep(kthread_t *);
static void     fss_tick(kthread_t *);
static void     fss_wakeup(kthread_t *);
static int      fss_donice(kthread_t *, cred_t *, int, int *);
static int      fss_doprio(kthread_t *, cred_t *, int, int *);
static pri_t    fss_globpri(kthread_t *);
static void     fss_yield(kthread_t *);
static void     fss_nullsys();

static struct classfuncs fss_classfuncs = {
        /* class functions */
        fss_admin,
        fss_getclinfo,
        fss_parmsin,
        fss_parmsout,
        fss_vaparmsin,
        fss_vaparmsout,
        fss_getclpri,
        fss_alloc,
        fss_free,

        /* thread functions */
        fss_enterclass,
        fss_exitclass,
        fss_canexit,
        fss_fork,
        fss_forkret,
        fss_parmsget,
        fss_parmsset,
        fss_stop,
        fss_exit,
        fss_active,
        fss_inactive,
        fss_swapin,
        fss_swapout,
        fss_trapret,
        fss_preempt,
        fss_setrun,
        fss_sleep,
        fss_tick,
        fss_wakeup,
        fss_donice,
        fss_globpri,
        fss_nullsys,    /* set_process_group */
        fss_yield,
        fss_doprio,
};
```
______***unchanged_portion_omitted_***

```
/*
 * fss_swapin() returns -1 if the thread is loaded or is not eligible to be
 * swapped in. Otherwise, it returns the thread's effective priority based
 * on swapout time and size of process (0 <= epri <= 0 SHRT_MAX).
 */
```

```
/*ARGSUSED*/
static pri_t
fss_swapin(kthread_t *t, int flags)
{
        fssproc_t *fssproc = FSSPROC(t);
        long epri = -1;
        proc_t *pp = ttoproc(t);

        ASSERT(THREAD_LOCK_HELD(t));

        if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
                time_t swapout_time;

                swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
                if (INHERITED(t) || (fssproc->fss_flags & FSSKPRI)) {
                        epri = (long)DISP_PRIO(t) + swapout_time;
                } else {
                        /*
                         * Threads which have been out for a long time,
                         * have high user mode priority and are associated
                         * with a small address space are more deserving.
                         */
                        epri = fssproc->fss_umdpri;
                        ASSERT(epri >= 0 && epri <= fss_maxumdpri);
                        epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
                }
                /*
                 * Scale epri so that SHRT_MAX / 2 represents zero priority.
                 */
                epri += SHRT_MAX / 2;
                if (epri < 0)
                        epri = 0;
                else if (epri > SHRT_MAX)
                        epri = SHRT_MAX;
        }
        return ((pri_t)epri);
}

/*
 * fss_swapout() returns -1 if the thread isn't loaded or is not eligible to
 * be swapped out. Otherwise, it returns the thread's effective priority
 * based on if the swapper is in softswap or hardswap mode.
 */
static pri_t
fss_swapout(kthread_t *t, int flags)
{
        fssproc_t *fssproc = FSSPROC(t);
        long epri = -1;
        proc_t *pp = ttoproc(t);
        time_t swapin_time;

        ASSERT(THREAD_LOCK_HELD(t));

        if (INHERITED(t) ||
            (fssproc->fss_flags & FSSKPRI) ||
            (t->t_proc_flag & TP_LWPEXIT) ||
            (t->t_state & (TS_ZOMB|TS_FREE|TS_STOPPED|TS_ONPROC|TS_WAIT)) ||
            !(t->t_schedflag & TS_LOAD) ||
            !(SWAP_OK(t)))
                return (-1);

        ASSERT(t->t_state & (TS_SLEEP | TS_RUN));

        swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;

        if (flags == SOFTSWAP) {
```

```
                if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
                        epri = 0;
                } else {
                        return ((pri_t)epri);
                }
        } else {
                pri_t pri;

                if ((t->t_state == TS_SLEEP && swapin_time > fss_minslp) ||
                    (t->t_state == TS_RUN && swapin_time > fss_minrun)) {
                        pri = fss_maxumdpri;
                        epri = swapin_time -
                            (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
                } else {
                        return ((pri_t)epri);
                }
        }

        /*
         * Scale epri so that SHRT_MAX / 2 represents zero priority.
         */
        epri += SHRT_MAX / 2;
        if (epri < 0)
                epri = 0;
        else if (epri > SHRT_MAX)
                epri = SHRT_MAX;

        return ((pri_t)epri);
}

/*
 * If thread is currently at a kernel mode priority (has slept) and is
 * returning to the userland we assign it the appropriate user mode priority
 * and time quantum here.  If we're lowering the thread's priority below that
 * of other runnable threads then we will set runrun via cpu_surrender() to
 * cause preemption.
 */
static void
fss_trapret(kthread_t *t)
{
        fssproc_t *fssproc = FSSPROC(t);
        cpu_t *cp = CPU;

        ASSERT(THREAD_LOCK_HELD(t));
        ASSERT(t == curthread);
        ASSERT(cp->cpu_dispthread == t);
        ASSERT(t->t_state == TS_ONPROC);

        t->t_kpri_req = 0;
        if (fssproc->fss_flags & FSSKPRI) {
                /*
                 * If thread has blocked in the kernel
                 */
                THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
                cp->cpu_dispatch_pri = DISP_PRIO(t);
                ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
                fssproc->fss_flags &= ~FSSKPRI;

                if (DISP_MUST_SURRENDER(t))
                        cpu_surrender(t);
        }

        /*
         * Swapout lwp if the swapper is waiting for this thread to reach
         * a safe point.
         */
        if (t->t_schedflag & TS_SWAPENQ) {
                thread_unlock(t);
                swapout_lwp(ttolwp(t));
                thread_lock(t);
        }
}

/*
 * Arrange for thread to be placed in appropriate location on dispatcher queue.
 * This is called with the current thread in TS_ONPROC and locked.
 */
static void
fss_preempt(kthread_t *t)
{
        fssproc_t *fssproc = FSSPROC(t);
        klwp_t *lwp;
        uint_t flags;

        ASSERT(t == curthread);
        ASSERT(THREAD_LOCK_HELD(curthread));
        ASSERT(t->t_state == TS_ONPROC);

        /*
         * If preempted in the kernel, make sure the thread has a kernel
         * priority if needed.
         */
        lwp = curthread->t_lwp;
        if (!(fssproc->fss_flags & FSSKPRI) && lwp != NULL && t->t_kpri_req) {
                fssproc->fss_flags |= FSSKPRI;
                THREAD_CHANGE_PRI(t, minclsyspri);
                ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
                t->t_trapret = 1;       /* so that fss_trapret will run */
                aston(t);
        }

        /*
         * This thread may be placed on wait queue by CPU Caps. In this case we
         * do not need to do anything until it is removed from the wait queue.
         * Do not enforce CPU caps on threads running at a kernel priority
         */
        if (CPUCAPS_ON()) {
                (void) cpucaps_charge(t, &fssproc->fss_caps,
                    CPUCAPS_CHARGE_ENFORCE);

                if (!(fssproc->fss_flags & FSSKPRI) && CPUCAPS_ENFORCE(t))
                        return;
        }

        /*
         * If preempted in user-land mark the thread as swappable because it
         * cannot be holding any kernel locks.
         */
        ASSERT(t->t_schedflag & TS_DONT_SWAP);
        if (lwp != NULL && lwp->lwp_state == LWP_USER)
                t->t_schedflag &= ~TS_DONT_SWAP;

        /*
         * Check to see if we're doing "preemption control" here.  If
         * we are, and if the user has requested that this thread not
         * be preempted, and if preemptions haven't been put off for
         * too long, let the preemption happen here but try to make
         * sure the thread is rescheduled as soon as possible.  We do
         * this by putting it on the front of the highest priority run
         * queue in the FSS class.  If the preemption has been put off
         * for too long, clear the "nopreempt" bit and let the thread
         * be preempted.
```

```
         */
        if (t->t_schedctl && schedctl_get_nopreempt(t)) {
                if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
                        DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
                        if (!(fssproc->fss_flags & FSSKPRI)) {
                                /*
                                 * If not already remembered, remember current
                                 * priority for restoration in fss_yield().
                                 */
                                if (!(fssproc->fss_flags & FSSRESTORE)) {
                                        fssproc->fss_scpri = t->t_pri;
                                        fssproc->fss_flags |= FSSRESTORE;
                                }
                                THREAD_CHANGE_PRI(t, fss_maxumdpri);
                                t->t_schedflag |= TS_DONT_SWAP;
                        }
                        schedctl_set_yield(t, 1);
                        setfrontdq(t);
                        return;
                } else {
                        if (fssproc->fss_flags & FSSRESTORE) {
                                THREAD_CHANGE_PRI(t, fssproc->fss_scpri);
                                fssproc->fss_flags &= ~FSSRESTORE;
                        }
                        schedctl_set_nopreempt(t, 0);
                        DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
                        /*
                         * Fall through and be preempted below.
                         */
                }
        }

        flags = fssproc->fss_flags & (FSSBACKQ | FSSKPRI);

        if (flags == FSSBACKQ) {
                fssproc->fss_timeleft = fss_quantum;
                fssproc->fss_flags &= ~FSSBACKQ;
                setbackdq(t);
        } else if (flags == (FSSBACKQ | FSSKPRI)) {
                fssproc->fss_flags &= ~FSSBACKQ;
                setbackdq(t);
        } else {
                setfrontdq(t);
        }
}
______unchanged_portion_omitted_

/*
 * Prepare thread for sleep. We reset the thread priority so it will run at the
 * kernel priority level when it wakes up.
 */
static void
fss_sleep(kthread_t *t)
{
        fssproc_t *fssproc = FSSPROC(t);

        ASSERT(t == curthread);
        ASSERT(THREAD_LOCK_HELD(t));

        ASSERT(t->t_state == TS_ONPROC);

        /*
         * Account for time spent on CPU before going to sleep.
         */
        (void) CPUCAPS_CHARGE(t, &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE);
        fss_inactive(t);

        /*
         * Assign a system priority to the thread and arrange for it to be
         * retained when the thread is next placed on the run queue (i.e.,
         * when it wakes up) instead of being given a new pri.  Also arrange
         * for trapret processing as the thread leaves the system call so it
         * will drop back to normal priority range.
         */
        if (t->t_kpri_req) {
                THREAD_CHANGE_PRI(t, minclsyspri);
                fssproc->fss_flags |= FSSKPRI;
                t->t_trapret = 1;       /* so that fss_trapret will run */
                aston(t);
        } else if (fssproc->fss_flags & FSSKPRI) {
                /*
                 * The thread has done a THREAD_KPRI_REQUEST(), slept, then
                 * done THREAD_KPRI_RELEASE() (so no t_kpri_req is 0 again),
                 * then slept again all without finishing the current system
                 * call so trapret won't have cleared FSSKPRI
                 */
                fssproc->fss_flags &= ~FSSKPRI;
                THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
                if (DISP_MUST_SURRENDER(curthread))
                        cpu_surrender(t);
        }
        t->t_stime = ddi_get_lbolt();   /* time stamp for the swapper */
}

/*
 * A tick interrupt has ocurrend on a running thread. Check to see if our
 * time slice has expired.
 * time slice has expired.  We must also clear the TS_DONT_SWAP flag in
 * t_schedflag if the thread is eligible to be swapped out.
 */
static void
fss_tick(kthread_t *t)
{
        fssproc_t *fssproc;
        fssproj_t *fssproj;
        klwp_t *lwp;
        boolean_t call_cpu_surrender = B_FALSE;
        boolean_t cpucaps_enforce = B_FALSE;

        ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

        /*
         * It's safe to access fsspset and fssproj structures because we're
         * holding our p_lock here.
         */
        thread_lock(t);
        fssproc = FSSPROC(t);
        fssproj = FSSPROC2FSSPROJ(fssproc);
        if (fssproj != NULL) {
                fsspset_t *fsspset = FSSPROJ2FSSPSET(fssproj);
                disp_lock_enter_high(&fsspset->fssps_displock);
                fssproj->fssp_ticks += fss_nice_tick[fssproc->fss_nice];
                fssproj->fssp_tick_cnt++;
                fssproc->fss_ticks++;
                disp_lock_exit_high(&fsspset->fssps_displock);
        }

        /*
         * Keep track of thread's project CPU usage.  Note that projects
         * get charged even when threads are running in the kernel.
         * Do not surrender CPU if running in the SYS class.
```

```
         */
        if (CPUCAPS_ON()) {
                cpucaps_enforce = cpucaps_charge(t,
                    &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE) &&
                    !(fssproc->fss_flags & FSSKPRI);
        }

        /*
         * A thread's execution time for threads running in the SYS class
         * is not tracked.
         */
        if ((fssproc->fss_flags & FSSKPRI) == 0) {
                /*
                 * If thread is not in kernel mode, decrement its fss_timeleft
                 */
                if (--fssproc->fss_timeleft <= 0) {
                        pri_t new_pri;

                        /*
                         * If we're doing preemption control and trying to
                         * avoid preempting this thread, just note that the
                         * thread should yield soon and let it keep running
                         * (unless it's been a while).
                         */
                        if (t->t_schedctl && schedctl_get_nopreempt(t)) {
                                if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
                                        DTRACE_SCHED1(schedctl__nopreempt,
                                            kthread_t *, t);
                                        schedctl_set_yield(t, 1);
                                        thread_unlock_nopreempt(t);
                                        return;
                                }
                        }
                        fssproc->fss_flags &= ~FSSRESTORE;

                        fss_newpri(fssproc, B_TRUE);
                        new_pri = fssproc->fss_umdpri;
                        ASSERT(new_pri >= 0 && new_pri <= fss_maxglobpri);

                        /*
                         * When the priority of a thread is changed, it may
                         * be necessary to adjust its position on a sleep queue
                         * or dispatch queue. The function thread_change_pri
                         * accomplishes this.
                         */
                        if (thread_change_pri(t, new_pri, 0)) {
                                if ((t->t_schedflag & TS_LOAD) &&
                                    (lwp = t->t_lwp) &&
                                    lwp->lwp_state == LWP_USER)
                                        t->t_schedflag &= ~TS_DONT_SWAP;
                                fssproc->fss_timeleft = fss_quantum;
                        } else {
                                call_cpu_surrender = B_TRUE;
                        }
                } else if (t->t_state == TS_ONPROC &&
                    t->t_pri < t->t_disp_queue->disp_maxrunpri) {
                        /*
                         * If there is a higher-priority thread which is
                         * waiting for a processor, then thread surrenders
                         * the processor.
                         */
                        call_cpu_surrender = B_TRUE;
                }
        }

        if (cpucaps_enforce && 2 * fssproc->fss_timeleft > fss_quantum) {
                /*
                 * The thread used more than half of its quantum, so assume that
                 * it used the whole quantum.
                 *
                 * Update thread's priority just before putting it on the wait
                 * queue so that it gets charged for the CPU time from its
                 * quantum even before that quantum expires.
                 */
                fss_newpri(fssproc, B_FALSE);
                if (t->t_pri != fssproc->fss_umdpri)
                        fss_change_priority(t, fssproc);

                /*
                 * We need to call cpu_surrender for this thread due to cpucaps
                 * enforcement, but fss_change_priority may have already done
                 * so. In this case FSSBACKQ is set and there is no need to call
                 * cpu-surrender again.
                 */
                if (!(fssproc->fss_flags & FSSBACKQ))
                        call_cpu_surrender = B_TRUE;
        }

        if (call_cpu_surrender) {
                fssproc->fss_flags |= FSSBACKQ;
                cpu_surrender(t);
        }

        thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
}

/*
 * Processes waking up go to the back of their queue.  We don't need to assign
 * a time quantum here because thread is still at a kernel mode priority and
 * the time slicing is not done for threads running in the kernel after
 * sleeping.  The proper time quantum will be assigned by fss_trapret before the
 * thread returns to user mode.
 */
static void
fss_wakeup(kthread_t *t)
{
        fssproc_t *fssproc;

        ASSERT(THREAD_LOCK_HELD(t));
        ASSERT(t->t_state == TS_SLEEP);

        fss_active(t);

        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
        fssproc = FSSPROC(t);
        fssproc->fss_flags &= ~FSSBACKQ;

        if (fssproc->fss_flags & FSSKPRI) {
                /*
                 * If we already have a kernel priority assigned, then we
                 * just use it.
                 */
                setbackdq(t);
        } else if (t->t_kpri_req) {
                /*
                 * Give thread a priority boost if we were asked.
                 */
                fssproc->fss_flags |= FSSKPRI;
                THREAD_CHANGE_PRI(t, minclsyspri);
                setbackdq(t);
                t->t_trapret = 1;       /* so that fss_trapret will run */
                aston(t);
```

```
        } else {
                /*
                 * Otherwise, we recalculate the priority.
                 */
                if (t->t_disp_time == ddi_get_lbolt()) {
                        setfrontdq(t);
                } else {
                        fssproc->fss_timeleft = fss_quantum;
                        THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
                        setbackdq(t);
                }
        }
}
______unchanged_portion_omitted_
```

****
42975 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/fx.c
6583 remove whole-process swapping
****
______*unchanged_portion_omitted_*

```
#define FX_ISVALID(pri, quantum) \
        (((pri >= 0) || (pri == FX_CB_NOCHANGE)) &&                     \
            ((quantum >= 0) || (quantum == FX_NOCHANGE) ||              \
                (quantum == FX_TQDEF) || (quantum == FX_TQINF)))


static id_t     fx_cid;         /* fixed priority class ID */
static fxdpent_t *fx_dptbl;     /* fixed priority disp parameter table */

static pri_t    fx_maxupri = FXMAXUPRI;
static pri_t    fx_maxumdpri;   /* max user mode fixed priority */

static pri_t    fx_maxglobpri;  /* maximum global priority used by fx class */
static kmutex_t fx_dptblock;    /* protects fixed priority dispatch table */


static kmutex_t fx_cb_list_lock[FX_CB_LISTS];   /* protects list of fxprocs */
                                                /* that have callbacks */
static fxproc_t fx_cb_plisthead[FX_CB_LISTS];   /* dummy fxproc at head of */
                                                /* list of fxprocs with */
                                                /* callbacks */

static int      fx_admin(caddr_t, cred_t *);
static int      fx_getclinfo(void *);
static int      fx_parmsin(void *);
static int      fx_parmsout(void *, pc_vaparms_t *);
static int      fx_vaparmsin(void *, pc_vaparms_t *);
static int      fx_vaparmsout(void *, pc_vaparms_t *);
static int      fx_getclpri(pcpri_t *);
static int      fx_alloc(void **, int);
static void     fx_free(void *);
static int      fx_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static void     fx_exitclass(void *);
static int      fx_canexit(kthread_t *, cred_t *);
static int      fx_fork(kthread_t *, kthread_t *, void *);
static void     fx_forkret(kthread_t *, kthread_t *);
static void     fx_parmsget(kthread_t *, void *);
static int      fx_parmsset(kthread_t *, void *, id_t, cred_t *);
static void     fx_stop(kthread_t *, int, int);
static void     fx_exit(kthread_t *);
static pri_t    fx_swapin(kthread_t *, int);
static pri_t    fx_swapout(kthread_t *, int);
static void     fx_trapret(kthread_t *);
static void     fx_preempt(kthread_t *);
static void     fx_setrun(kthread_t *);
static void     fx_sleep(kthread_t *);
static void     fx_tick(kthread_t *);
static void     fx_wakeup(kthread_t *);
static int      fx_donice(kthread_t *, cred_t *, int, int *);
static int      fx_doprio(kthread_t *, cred_t *, int, int *);
static pri_t    fx_globpri(kthread_t *);
static void     fx_yield(kthread_t *);
static void     fx_nullsys();

extern fxdpent_t *fx_getdptbl(void);

static void     fx_change_priority(kthread_t *, fxproc_t *);
static fxproc_t *fx_list_lookup(kt_did_t);
static void fx_list_release(fxproc_t *);
```

```
static struct classfuncs fx_classfuncs = {
        /* class functions */
        fx_admin,
        fx_getclinfo,
        fx_parmsin,
        fx_parmsout,
        fx_vaparmsin,
        fx_vaparmsout,
        fx_getclpri,
        fx_alloc,
        fx_free,

        /* thread functions */
        fx_enterclass,
        fx_exitclass,
        fx_canexit,
        fx_fork,
        fx_forkret,
        fx_parmsget,
        fx_parmsset,
        fx_stop,
        fx_exit,
        fx_nullsys,     /* active */
        fx_nullsys,     /* inactive */
        fx_swapin,
        fx_swapout,
        fx_trapret,
        fx_preempt,
        fx_setrun,
        fx_sleep,
        fx_tick,
        fx_wakeup,
        fx_donice,
        fx_globpri,
        fx_nullsys,     /* set_process_group */
        fx_yield,
        fx_doprio,
};
```
______*unchanged_portion_omitted_*

```
/*
 * Prepare thread for sleep. We reset the thread priority so it will
 * run at the kernel priority level when it wakes up.
 */
static void
fx_sleep(kthread_t *t)
{
        fxproc_t        *fxpp = (fxproc_t *)(t->t_cldata);

        ASSERT(t == curthread);
        ASSERT(THREAD_LOCK_HELD(t));

        /*
         * Account for time spent on CPU before going to sleep.
         */
        (void) CPUCAPS_CHARGE(t, &fxpp->fx_caps, CPUCAPS_CHARGE_ENFORCE);

        if (FX_HAS_CB(fxpp)) {
                FX_CB_SLEEP(FX_CALLB(fxpp), fxpp->fx_cookie);
        }
        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
}
```

```
/*
 * Return Values:
 *
 *      -1 if the thread is loaded or is not eligible to be swapped in.
 *
 * FX and RT threads are designed so that they don't swapout; however,
 * it is possible that while the thread is swapped out and in another class, it
 * can be changed to FX or RT.  Since these threads should be swapped in
 * as soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
 * returns SHRT_MAX - 1, so that it gives deference to any swapped out
 * RT threads.
 */
/* ARGSUSED */
static pri_t
fx_swapin(kthread_t *t, int flags)
{
        pri_t   tpri = -1;

        ASSERT(THREAD_LOCK_HELD(t));

        if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
                tpri = (pri_t)SHRT_MAX - 1;
        }

        return (tpri);
}

/*
 * Return Values
 *      -1 if the thread isn't loaded or is not eligible to be swapped out.
 */
/* ARGSUSED */
static pri_t
fx_swapout(kthread_t *t, int flags)
{
        ASSERT(THREAD_LOCK_HELD(t));

        return (-1);

}
```
______***unchanged_portion_omitted_***

```
/*
 * Processes waking up go to the back of their queue.
 */
static void
fx_wakeup(kthread_t *t)
{
        fxproc_t        *fxpp = (fxproc_t *)(t->t_cldata);

        ASSERT(THREAD_LOCK_HELD(t));

        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
        if (FX_HAS_CB(fxpp)) {
                clock_t new_quantum =  (clock_t)fxpp->fx_pquantum;
                pri_t   newpri = fxpp->fx_pri;
                FX_CB_WAKEUP(FX_CALLB(fxpp), fxpp->fx_cookie,
                    &new_quantum, &newpri);
                FX_ADJUST_QUANTUM(new_quantum);
                if ((int)new_quantum != fxpp->fx_pquantum) {
                        fxpp->fx_pquantum = (int)new_quantum;
                        fxpp->fx_timeleft = fxpp->fx_pquantum;
                }
```

```
                FX_ADJUST_PRI(newpri);
                if (newpri != fxpp->fx_pri) {
                        fxpp->fx_pri = newpri;
                        THREAD_CHANGE_PRI(t, fx_dptbl[fxpp->fx_pri].fx_globpri);
                }
        }

        fxpp->fx_flags &= ~FXBACKQ;

        if (t->t_disp_time != ddi_get_lbolt())
                setbackdq(t);
        else
                setfrontdq(t);
}
```
______***unchanged_portion_omitted_***

****
25930 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/rt.c
6583 remove whole-process swapping
****
______**unchanged_portion_omitted_**

```
/*
 * Class specific code for the real-time class
 */

/*
 * Extern declarations for variables defined in the rt master file
 */
#define RTMAXPRI 59

pri_t rt_maxpri = RTMAXPRI;     /* maximum real-time priority */
rtdpent_t *rt_dptbl;      /* real-time dispatcher parameter table */

/*
 * control flags (kparms->rt_cflags).
 */
#define RT_DOPRI        0x01    /* change priority */
#define RT_DOTQ         0x02    /* change RT time quantum */
#define RT_DOSIG        0x04    /* change RT time quantum signal */

static int      rt_admin(caddr_t, cred_t *);
static int      rt_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static int      rt_fork(kthread_t *, kthread_t *, void *);
static int      rt_getclinfo(void *);
static int      rt_getclpri(pcpri_t *);
static int      rt_parmsin(void *);
static int      rt_parmsout(void *, pc_vaparms_t *);
static int      rt_vaparmsin(void *, pc_vaparms_t *);
static int      rt_vaparmsout(void *, pc_vaparms_t *);
static int      rt_parmsset(kthread_t *, void *, id_t, cred_t *);
static int      rt_donice(kthread_t *, cred_t *, int, int *);
static int      rt_doprio(kthread_t *, cred_t *, int, int *);
static void     rt_exitclass(void *);
static int      rt_canexit(kthread_t *, cred_t *);
static void     rt_forkret(kthread_t *, kthread_t *);
static void     rt_nullsys();
static void     rt_parmsget(kthread_t *, void *);
static void     rt_preempt(kthread_t *);
static void     rt_setrun(kthread_t *);
static void     rt_tick(kthread_t *);
static void     rt_wakeup(kthread_t *);
static pri_t    rt_swapin(kthread_t *, int);
static pri_t    rt_swapout(kthread_t *, int);
static pri_t    rt_globpri(kthread_t *);
static void     rt_yield(kthread_t *);
static int      rt_alloc(void **, int);
static void     rt_free(void *);

static void     rt_change_priority(kthread_t *, rtproc_t *);

static id_t     rt_cid;         /* real-time class ID */
static rtproc_t rt_plisthead;   /* dummy rtproc at head of rtproc list */
static kmutex_t rt_dptblock;    /* protects realtime dispatch table */
static kmutex_t rt_list_lock;   /* protects RT thread list */

extern rtdpent_t *rt_getdptbl(void);

static struct classfuncs rt_classfuncs = {
        /* class ops */
```

```
        rt_admin,
        rt_getclinfo,
        rt_parmsin,
        rt_parmsout,
        rt_vaparmsin,
        rt_vaparmsout,
        rt_getclpri,
        rt_alloc,
        rt_free,
        /* thread ops */
        rt_enterclass,
        rt_exitclass,
        rt_canexit,
        rt_fork,
        rt_forkret,
        rt_parmsget,
        rt_parmsset,
        rt_nullsys,     /* stop */
        rt_nullsys,     /* exit */
        rt_nullsys,     /* active */
        rt_nullsys,     /* inactive */
        rt_swapin,
        rt_swapout,
        rt_nullsys,     /* trapret */
        rt_preempt,
        rt_setrun,
        rt_nullsys,     /* sleep */
        rt_tick,
        rt_wakeup,
        rt_donice,
        rt_globpri,
        rt_nullsys,     /* set_process_group */
        rt_yield,
        rt_doprio,
};
```
______**unchanged_portion_omitted_**

```
/*
 * Arrange for thread to be placed in appropriate location
 * on dispatcher queue.  Runs at splhi() since the clock
 * interrupt can cause RTBACKQ to be set.
 */
static void
rt_preempt(kthread_t *t)
{
        rtproc_t *rtpp = (rtproc_t *)(t->t_cldata);
        klwp_t *lwp;

        ASSERT(THREAD_LOCK_HELD(t));

        /*
         * If the state is user I allow swapping because I know I won't
         * be holding any locks.
         */
        if ((lwp = curthread->t_lwp) != NULL && lwp->lwp_state == LWP_USER)
                t->t_schedflag &= ~TS_DONT_SWAP;
        if ((rtpp->rt_flags & RTBACKQ) != 0) {
                rtpp->rt_timeleft = rtpp->rt_pquantum;
                rtpp->rt_flags &= ~RTBACKQ;
                setbackdq(t);
        } else
                setfrontdq(t);

}
```
______**unchanged_portion_omitted_**

```
static void
rt_setrun(kthread_t *t)
{
        rtproc_t *rtpp = (rtproc_t *)(t->t_cldata);

        ASSERT(THREAD_LOCK_HELD(t));

        rtpp->rt_timeleft = rtpp->rt_pquantum;
        rtpp->rt_flags &= ~RTBACKQ;
        setbackdq(t);
}

/*
 * Returns the priority of the thread, -1 if the thread is loaded or ineligible
 * for swapin.
 *
 * FX and RT threads are designed so that they don't swapout; however, it
 * is possible that while the thread is swapped out and in another class, it
 * can be changed to FX or RT.  Since these threads should be swapped in as
 * soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
 * returns SHRT_MAX - 1, so that it gives deference to any swapped out RT
 * threads.
 */
/* ARGSUSED */
static pri_t
rt_swapin(kthread_t *t, int flags)
{
        pri_t   tpri = -1;

        ASSERT(THREAD_LOCK_HELD(t));

        if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
                tpri = (pri_t)SHRT_MAX;
        }

        return (tpri);
}

/*
 * Return an effective priority for swapout.
 */
/* ARGSUSED */
static pri_t
rt_swapout(kthread_t *t, int flags)
{
        ASSERT(THREAD_LOCK_HELD(t));

        return (-1);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
    4804 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/sysclass.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2008 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"        /* from SVr4.0 1.12 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/signal.h>
#include <sys/pcb.h>
#include <sys/user.h>
#include <sys/systm.h>
#include <sys/sysinfo.h>
#include <sys/var.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/proc.h>
#include <sys/debug.h>
#include <sys/inline.h>
#include <sys/disp.h>
#include <sys/class.h>
#include <sys/kmem.h>
#include <sys/cpuvar.h>
#include <sys/priocntl.h>

/*
 * Class specific code for the sys class. There are no
 * class specific data structures associated with
 * the sys class and the scheduling policy is trivially
 * simple. There is no time slicing.
 */

pri_t           sys_init(id_t, int, classfuncs_t **);
static int      sys_getclpri(pcpri_t *);
static int      sys_fork(kthread_t *, kthread_t *, void *);
```

```
static int      sys_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static int      sys_canexit(kthread_t *, cred_t *);
static int      sys_nosys();
static int      sys_donice(kthread_t *, cred_t *, int, int *);
static int      sys_doprio(kthread_t *, cred_t *, int, int *);
static void     sys_forkret(kthread_t *, kthread_t *);
static void     sys_nullsys();
static pri_t    sys_swappri(kthread_t *, int);
static int      sys_alloc(void **, int);

struct classfuncs sys_classfuncs = {
        /* messages to class manager */
        {
                sys_nosys,      /* admin */
                sys_nosys,      /* getclinfo */
                sys_nosys,      /* parmsin */
                sys_nosys,      /* parmsout */
                sys_nosys,      /* vaparmsin */
                sys_nosys,      /* vaparmsout */
                sys_getclpri,   /* getclpri */
                sys_alloc,
                sys_nullsys,    /* free */
        },
        /* operations on threads */
        {
                sys_enterclass, /* enterclass */
                sys_nullsys,    /* exitclass */
                sys_canexit,
                sys_fork,
                sys_forkret,    /* forkret */
                sys_nullsys,    /* parmsget */
                sys_nosys,      /* parmsset */
                sys_nullsys,    /* stop */
                sys_nullsys,    /* exit */
                sys_nullsys,    /* active */
                sys_nullsys,    /* inactive */
                sys_swappri,    /* swapin */
                sys_swappri,    /* swapout */
                sys_nullsys,    /* trapret */
                setfrontdq,     /* preempt */
                setbackdq,      /* setrun */
                sys_nullsys,    /* sleep */
                sys_nullsys,    /* tick */
                setbackdq,      /* wakeup */
                sys_donice,
                (pri_t (*)())sys_nosys, /* globpri */
                sys_nullsys,    /* set_process_group */
                sys_nullsys,    /* yield */
                sys_doprio,
        }

};
______unchanged_portion_omitted_


/* ARGSUSED */
static void
sys_forkret(t, ct)
        kthread_t *t;
        kthread_t *ct;
{
        register proc_t *pp = ttoproc(t);
        register proc_t *cp = ttoproc(ct);

        ASSERT(t == curthread);
        ASSERT(MUTEX_HELD(&pidlock));
```

```
	/*
	 * Grab the child's p_lock before dropping pidlock to ensure
	 * the process does not disappear before we set it running.
	 */
	mutex_enter(&cp->p_lock);
	mutex_exit(&pidlock);
	continuelwps(cp);
	mutex_exit(&cp->p_lock);

	mutex_enter(&pp->p_lock);
	continuelwps(pp);
	mutex_exit(&pp->p_lock);
}

/* ARGSUSED */
static pri_t
sys_swappri(t, flags)
	kthread_t	*t;
	int		flags;
{
	return (-1);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   37694 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/sysdc.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*ARGSUSED*/
static pri_t
sysdc_no_swap(kthread_t *t, int flags)
{
        /* SDC threads cannot be swapped. */
        return (-1);
}

/*
 * Get maximum and minimum priorities enjoyed by SDC threads.
 */
static int
sysdc_getclpri(pcpri_t *pcprip)
{
        pcprip->pc_clpmax = sysdc_maxpri;
        pcprip->pc_clpmin = sysdc_minpri;
        return (0);
}
______unchanged_portion_omitted_

static int sysdc_enosys();      /* Boy, ANSI-C's K&R compatibility is weird. */
static int sysdc_einval();
static void sysdc_nullsys();

static struct classfuncs sysdc_classfuncs = {
        /* messages to class manager */
        {
                sysdc_enosys,   /* admin */
                sysdc_getclinfo,
                sysdc_enosys,   /* parmsin */
                sysdc_enosys,   /* parmsout */
                sysdc_enosys,   /* vaparmsin */
                sysdc_enosys,   /* vaparmsout */
                sysdc_getclpri,
                sysdc_alloc,
                sysdc_free,
        },
        /* operations on threads */
        {
                sysdc_enterclass,
                sysdc_exitclass,
                sysdc_canexit,
                sysdc_fork,
                sysdc_forkret,
                sysdc_nullsys,  /* parmsget */
                sysdc_enosys,   /* parmsset */
                sysdc_nullsys,  /* stop */
                sysdc_exit,
                sysdc_nullsys,  /* active */
                sysdc_nullsys,  /* inactive */
                sysdc_no_swap,  /* swapin */
                sysdc_no_swap,  /* swapout */
                sysdc_nullsys,  /* trapret */
                sysdc_preempt,
                sysdc_setrun,
                sysdc_sleep,
                sysdc_tick,
                sysdc_wakeup,
                sysdc_einval,   /* donice */
```

```
                sysdc_globpri,
                sysdc_nullsys,  /* set_process_group */
                sysdc_nullsys,  /* yield */
                sysdc_einval,   /* doprio */
        }
};
______unchanged_portion_omitted_
```

```
************************************************************
   53361 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/thread.c
6583 remove whole-process swapping
************************************************************
______unchanged_portion_omitted_

/*
 * Create a thread.
 *
 * thread_create() blocks for memory if necessary.  It never fails.
 *
 * If stk is NULL, the thread is created at the base of the stack
 * and cannot be swapped.
 */
kthread_t *
thread_create(
        caddr_t stk,
        size_t  stksize,
        void    (*proc)(),
        void    *arg,
        size_t  len,
        proc_t   *pp,
        int     state,
        pri_t   pri)
{
        kthread_t *t;
        extern struct classfuncs sys_classfuncs;
        turnstile_t *ts;

        /*
         * Every thread keeps a turnstile around in case it needs to block.
         * The only reason the turnstile is not simply part of the thread
         * structure is that we may have to break the association whenever
         * more than one thread blocks on a given synchronization object.
         * From a memory-management standpoint, turnstiles are like the
         * "attached mblks" that hang off dblks in the streams allocator.
         */
        ts = kmem_cache_alloc(turnstile_cache, KM_SLEEP);

        if (stk == NULL) {
                /*
                 * alloc both thread and stack in segkp chunk
                 */

                if (stksize < default_stksize)
                        stksize = default_stksize;

                if (stksize == default_stksize) {
                        stk = (caddr_t)segkp_cache_get(segkp_thread);
                } else {
                        stksize = roundup(stksize, PAGESIZE);
                        stk = (caddr_t)segkp_get(segkp, stksize,
                            (KPD_HASREDZONE | KPD_NO_ANON | KPD_LOCKED));
                }

                ASSERT(stk != NULL);

                /*
                 * The machine-dependent mutex code may require that
                 * thread pointers (since they may be used for mutex owner
                 * fields) have certain alignment requirements.
                 * PTR24_ALIGN is the size of the alignment quanta.
                 * XXX - assumes stack grows toward low addresses.
                 */
                if (stksize <= sizeof (kthread_t) + PTR24_ALIGN)
```

```
                        cmn_err(CE_PANIC, "thread_create: proposed stack size"
                            " too small to hold thread.");
#ifdef STACK_GROWTH_DOWN
                stksize -= SA(sizeof (kthread_t) + PTR24_ALIGN - 1);
                stksize &= -PTR24_ALIGN;        /* make thread aligned */
                t = (kthread_t *)(stk + stksize);
                bzero(t, sizeof (kthread_t));
                if (audit_active)
                        audit_thread_create(t);
                t->t_stk = stk + stksize;
                t->t_stkbase = stk;
#else   /* stack grows to larger addresses */
                stksize -= SA(sizeof (kthread_t));
                t = (kthread_t *)(stk);
                bzero(t, sizeof (kthread_t));
                t->t_stk = stk + sizeof (kthread_t);
                t->t_stkbase = stk + stksize + sizeof (kthread_t);
#endif  /* STACK_GROWTH_DOWN */
                t->t_flag |= T_TALLOCSTK;
                t->t_swap = stk;
        } else {
                t = kmem_cache_alloc(thread_cache, KM_SLEEP);
                bzero(t, sizeof (kthread_t));
                ASSERT(((uintptr_t)t & (PTR24_ALIGN - 1)) == 0);
                if (audit_active)
                        audit_thread_create(t);
                /*
                 * Initialize t_stk to the kernel stack pointer to use
                 * upon entry to the kernel
                 */
#ifdef STACK_GROWTH_DOWN
                t->t_stk = stk + stksize;
                t->t_stkbase = stk;
#else
                t->t_stk = stk;                 /* 3b2-like */
                t->t_stkbase = stk + stksize;
#endif /* STACK_GROWTH_DOWN */
        }

        if (kmem_stackinfo != 0) {
                stkinfo_begin(t);
        }

        t->t_ts = ts;

        /*
         * p_cred could be NULL if it thread_create is called before cred_init
         * is called in main.
         */
        mutex_enter(&pp->p_crlock);
        if (pp->p_cred)
                crhold(t->t_cred = pp->p_cred);
        mutex_exit(&pp->p_crlock);
        t->t_start = gethrestime_sec();
        t->t_startpc = proc;
        t->t_procp = pp;
        t->t_clfuncs = &sys_classfuncs.thread;
        t->t_cid = syscid;
        t->t_pri = pri;
        t->t_schedflag = 0;
        t->t_stime = ddi_get_lbolt();
        t->t_schedflag = TS_LOAD | TS_DONT_SWAP;
        t->t_bind_cpu = PBIND_NONE;
        t->t_bindflag = (uchar_t)default_binding_mode;
        t->t_bind_pset = PS_NONE;
        t->t_plockp = &pp->p_lock;
```

```
	t->t_copyops = NULL;
	t->t_taskq = NULL;
	t->t_anttime = 0;
	t->t_hatdepth = 0;

	t->t_dtrace_vtime = 1;	/* assure vtimestamp is always non-zero */

	CPU_STATS_ADDQ(CPU, sys, nthreads, 1);
#ifndef NPROBE
	/* Kernel probe */
	tnf_thread_create(t);
#endif /* NPROBE */
	LOCK_INIT_CLEAR(&t->t_lock);

	/*
	 * Callers who give us a NULL proc must do their own
	 * stack initialization.  e.g. lwp_create()
	 */
	if (proc != NULL) {
		t->t_stk = thread_stk_init(t->t_stk);
		thread_load(t, proc, arg, len);
	}

	/*
	 * Put a hold on project0. If this thread is actually in a
	 * different project, then t_proj will be changed later in
	 * lwp_create().  All kernel-only threads must be in project 0.
	 */
	t->t_proj = project_hold(proj0p);

	lgrp_affinity_init(&t->t_lgrp_affinity);

	mutex_enter(&pidlock);
	nthread++;
	t->t_did = next_t_id++;
	t->t_prev = curthread->t_prev;
	t->t_next = curthread;

	/*
	 * Add the thread to the list of all threads, and initialize
	 * its t_cpu pointer.  We need to block preemption since
	 * cpu_offline walks the thread list looking for threads
	 * with t_cpu pointing to the CPU being offlined.  We want
	 * to make sure that the list is consistent and that if t_cpu
	 * is set, the thread is on the list.
	 */
	kpreempt_disable();
	curthread->t_prev->t_next = t;
	curthread->t_prev = t;

	/*
	 * Threads should never have a NULL t_cpu pointer so assign it
	 * here.  If the thread is being created with state TS_RUN a
	 * better CPU may be chosen when it is placed on the run queue.
	 *
	 * We need to keep kernel preemption disabled when setting all
	 * three fields to keep them in sync.  Also, always create in
	 * the default partition since that's where kernel threads go
	 * (if this isn't a kernel thread, t_cpupart will be changed
	 * in lwp_create before setting the thread runnable).
	 */
	t->t_cpupart = &cp_default;

	/*
	 * For now, affiliate this thread with the root lgroup.
	 * Since the kernel does not (presently) allocate its memory
```

```
	 * in a locality aware fashion, the root is an appropriate home.
	 * If this thread is later associated with an lwp, it will have
	 * it's lgroup re-assigned at that time.
	 */
	lgrp_move_thread(t, &cp_default.cp_lgrploads[LGRP_ROOTID], 1);

	/*
	 * Inherit the current cpu.  If this cpu isn't part of the chosen
	 * lgroup, a new cpu will be chosen by cpu_choose when the thread
	 * is ready to run.
	 */
	if (CPU->cpu_part == &cp_default)
		t->t_cpu = CPU;
	else
		t->t_cpu = disp_lowpri_cpu(cp_default.cp_cpulist, t->t_lpl,
		    t->t_pri, NULL);

	t->t_disp_queue = t->t_cpu->cpu_disp;
	kpreempt_enable();

	/*
	 * Initialize thread state and the dispatcher lock pointer.
	 * Need to hold onto pidlock to block allthreads walkers until
	 * the state is set.
	 */
	switch (state) {
	case TS_RUN:
		curthread->t_oldspl = splhigh();	/* get dispatcher spl */
		THREAD_SET_STATE(t, TS_STOPPED, &transition_lock);
		CL_SETRUN(t);
		thread_unlock(t);
		break;

	case TS_ONPROC:
		THREAD_ONPROC(t, t->t_cpu);
		break;

	case TS_FREE:
		/*
		 * Free state will be used for intr threads.
		 * The interrupt routine must set the thread dispatcher
		 * lock pointer (t_lockp) if starting on a CPU
		 * other than the current one.
		 */
		THREAD_FREEINTR(t, CPU);
		break;

	case TS_STOPPED:
		THREAD_SET_STATE(t, TS_STOPPED, &stop_lock);
		break;

	default:			/* TS_SLEEP, TS_ZOMB or TS_TRANS */
		cmn_err(CE_PANIC, "thread_create: invalid state %d", state);
	}
	mutex_exit(&pidlock);
	return (t);
}
```
______***unchanged_portion_omitted_***

****
   57791 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/ts.c
6583 remove whole-process swapping
****
______*unchanged_portion_omitted_*

```
static int      ts_admin(caddr_t, cred_t *);
static int      ts_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static int      ts_fork(kthread_t *, kthread_t *, void *);
static int      ts_getclinfo(void *);
static int      ts_getclpri(pcpri_t *);
static int      ts_parmsin(void *);
static int      ts_parmsout(void *, pc_vaparms_t *);
static int      ts_vaparmsin(void *, pc_vaparms_t *);
static int      ts_vaparmsout(void *, pc_vaparms_t *);
static int      ts_parmsset(kthread_t *, void *, id_t, cred_t *);
static void     ts_exit(kthread_t *);
static int      ts_donice(kthread_t *, cred_t *, int, int *);
static int      ts_doprio(kthread_t *, cred_t *, int, int *);
static void     ts_exitclass(void *);
static int      ts_canexit(kthread_t *, cred_t *);
static void     ts_forkret(kthread_t *, kthread_t *);
static void     ts_nullsys();
static void     ts_parmsget(kthread_t *, void *);
static void     ts_preempt(kthread_t *);
static void     ts_setrun(kthread_t *);
static void     ts_sleep(kthread_t *);
static pri_t    ts_swapin(kthread_t *, int);
static pri_t    ts_swapout(kthread_t *, int);
static void     ts_tick(kthread_t *);
static void     ts_trapret(kthread_t *);
static void     ts_update(void *);
static int      ts_update_list(int);
static void     ts_wakeup(kthread_t *);
static pri_t    ts_globpri(kthread_t *);
static void     ts_yield(kthread_t *);
extern tsdpent_t *ts_getdptbl(void);
extern pri_t    *ts_getkmdpris(void);
extern pri_t    td_getmaxumdpri(void);
static int      ts_alloc(void **, int);
static void     ts_free(void *);

pri_t           ia_init(id_t, int, classfuncs_t **);
static int      ia_getclinfo(void *);
static int      ia_getclpri(pcpri_t *);
static int      ia_parmsin(void *);
static int      ia_vaparmsin(void *, pc_vaparms_t *);
static int      ia_vaparmsout(void *, pc_vaparms_t *);
static int      ia_parmsset(kthread_t *, void *, id_t, cred_t *);
static void     ia_parmsget(kthread_t *, void *);
static void     ia_set_process_group(pid_t, pid_t, pid_t);

static void     ts_change_priority(kthread_t *, tsproc_t *);

extern pri_t    ts_maxkmdpri;   /* maximum kernel mode ts priority */
static pri_t    ts_maxglobpri;  /* maximum global priority used by ts class */
static kmutex_t ts_dptblock;    /* protects time sharing dispatch table */
static kmutex_t ts_list_lock[TS_LISTS]; /* protects tsproc lists */
static tsproc_t ts_plisthead[TS_LISTS]; /* dummy tsproc at head of lists */

static gid_t    IA_gid = 0;

static struct classfuncs ts_classfuncs = {
        /* class functions */
```

```
        ts_admin,
        ts_getclinfo,
        ts_parmsin,
        ts_parmsout,
        ts_vaparmsin,
        ts_vaparmsout,
        ts_getclpri,
        ts_alloc,
        ts_free,

        /* thread functions */
        ts_enterclass,
        ts_exitclass,
        ts_canexit,
        ts_fork,
        ts_forkret,
        ts_parmsget,
        ts_parmsset,
        ts_nullsys,     /* stop */
        ts_exit,
        ts_nullsys,     /* active */
        ts_nullsys,     /* inactive */
        ts_swapin,
        ts_swapout,
        ts_trapret,
        ts_preempt,
        ts_setrun,
        ts_sleep,
        ts_tick,
        ts_wakeup,
        ts_donice,
        ts_globpri,
        ts_nullsys,     /* set_process_group */
        ts_yield,
        ts_doprio,
};

/*
 * ia_classfuncs is used for interactive class threads; IA threads are stored
 * on the same class list as TS threads, and most of the class functions are
 * identical, but a few have different enough functionality to require their
 * own functions.
 */
static struct classfuncs ia_classfuncs = {
        /* class functions */
        ts_admin,
        ia_getclinfo,
        ia_parmsin,
        ts_parmsout,
        ia_vaparmsin,
        ia_vaparmsout,
        ia_getclpri,
        ts_alloc,
        ts_free,

        /* thread functions */
        ts_enterclass,
        ts_exitclass,
        ts_canexit,
        ts_fork,
        ts_forkret,
        ia_parmsget,
        ia_parmsset,
        ts_nullsys,     /* stop */
        ts_exit,
        ts_nullsys,     /* active */
        ts_nullsys,     /* inactive */
        ts_swapin,
        ts_swapout,
        ts_trapret,
        ts_preempt,
        ts_setrun,
        ts_sleep,
        ts_tick,
        ts_wakeup,
        ts_donice,
        ts_globpri,
        ia_set_process_group,
        ts_yield,
        ts_doprio,
};
```

______***unchanged_portion_omitted_***

```
/*
 * Arrange for thread to be placed in appropriate location
 * on dispatcher queue.
 *
 * This is called with the current thread in TS_ONPROC and locked.
 */
static void
ts_preempt(kthread_t *t)
{
        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
        klwp_t          *lwp = curthread->t_lwp;
        pri_t           oldpri = t->t_pri;

        ASSERT(t == curthread);
        ASSERT(THREAD_LOCK_HELD(curthread));

        /*
         * If preempted in the kernel, make sure the thread has
         * a kernel priority if needed.
         */
        if (!(tspp->ts_flags & TSKPRI) && lwp != NULL && t->t_kpri_req) {
                tspp->ts_flags |= TSKPRI;
                THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
                t->t_trapret = 1;               /* so ts_trapret will run */
                aston(t);
        }

        /*
         * This thread may be placed on wait queue by CPU Caps. In this case we
         * do not need to do anything until it is removed from the wait queue.
         * Do not enforce CPU caps on threads running at a kernel priority
         */
        if (CPUCAPS_ON()) {
                (void) cpucaps_charge(t, &tspp->ts_caps,
                    CPUCAPS_CHARGE_ENFORCE);
                if (!(tspp->ts_flags & TSKPRI) && CPUCAPS_ENFORCE(t))
                        return;
        }

        /*
         * If thread got preempted in the user-land then we know
         * it isn't holding any locks.  Mark it as swappable.
         */
        ASSERT(t->t_schedflag & TS_DONT_SWAP);
        if (lwp != NULL && lwp->lwp_state == LWP_USER)
                t->t_schedflag &= ~TS_DONT_SWAP;

        /*
```

```
         * Check to see if we're doing "preemption control" here.  If
         * we are, and if the user has requested that this thread not
         * be preempted, and if preemptions haven't been put off for
         * too long, let the preemption happen here but try to make
         * sure the thread is rescheduled as soon as possible.  We do
         * this by putting it on the front of the highest priority run
         * queue in the TS class.  If the preemption has been put off
         * for too long, clear the "nopreempt" bit and let the thread
         * be preempted.
         */
        if (t->t_schedctl && schedctl_get_nopreempt(t)) {
                if (tspp->ts_timeleft > -SC_MAX_TICKS) {
                        DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
                        if (!(tspp->ts_flags & TSKPRI)) {
                                /*
                                 * If not already remembered, remember current
                                 * priority for restoration in ts_yield().
                                 */
                                if (!(tspp->ts_flags & TSRESTORE)) {
                                        tspp->ts_scpri = t->t_pri;
                                        tspp->ts_flags |= TSRESTORE;
                                }
                                THREAD_CHANGE_PRI(t, ts_maxumdpri);
                                t->t_schedflag |= TS_DONT_SWAP;
                        }
                        schedctl_set_yield(t, 1);
                        setfrontdq(t);
                        goto done;
                } else {
                        if (tspp->ts_flags & TSRESTORE) {
                                THREAD_CHANGE_PRI(t, tspp->ts_scpri);
                                tspp->ts_flags &= ~TSRESTORE;
                        }
                        schedctl_set_nopreempt(t, 0);
                        DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
                        TNF_PROBE_2(schedctl_preempt, "schedctl TS ts_preempt",
                            /* CSTYLED */, tnf_pid, pid, ttoproc(t)->p_pid,
                            tnf_lwpid, lwpid, t->t_tid);
                        /*
                         * Fall through and be preempted below.
                         */
                }
        }

        if ((tspp->ts_flags & (TSBACKQ|TSKPRI)) == TSBACKQ) {
                tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
                tspp->ts_dispwait = 0;
                tspp->ts_flags &= ~TSBACKQ;
                setbackdq(t);
        } else if ((tspp->ts_flags & (TSBACKQ|TSKPRI)) == (TSBACKQ|TSKPRI)) {
                tspp->ts_flags &= ~TSBACKQ;
                setbackdq(t);
        } else {
                setfrontdq(t);
        }

done:
        TRACE_2(TR_FAC_DISP, TR_PREEMPT,
            "preempt:tid %p old pri %d", t, oldpri);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Prepare thread for sleep. We reset the thread priority so it will
 * run at the kernel priority level when it wakes up.
 */
static void
ts_sleep(kthread_t *t)
{
        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
        int             flags;
        pri_t           old_pri = t->t_pri;

        ASSERT(t == curthread);
        ASSERT(THREAD_LOCK_HELD(t));

        /*
         * Account for time spent on CPU before going to sleep.
         */
        (void) CPUCAPS_CHARGE(t, &tspp->ts_caps, CPUCAPS_CHARGE_ENFORCE);

        flags = tspp->ts_flags;
        if (t->t_kpri_req) {
                tspp->ts_flags = flags | TSKPRI;
                THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
                t->t_trapret = 1;               /* so ts_trapret will run */
                aston(t);
        } else if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
                /*
                 * If thread has blocked in the kernel (as opposed to
                 * being merely preempted), recompute the user mode priority.
                 */
                tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
                TS_NEWUMDPRI(tspp);
                tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
                tspp->ts_dispwait = 0;

                THREAD_CHANGE_PRI(curthread,
                    ts_dptbl[tspp->ts_umdpri].ts_globpri);
                ASSERT(curthread->t_pri >= 0 &&
                    curthread->t_pri <= ts_maxglobpri);
                tspp->ts_flags = flags & ~TSKPRI;

                if (DISP_MUST_SURRENDER(curthread))
                        cpu_surrender(curthread);
        } else if (flags & TSKPRI) {
                THREAD_CHANGE_PRI(curthread,
                    ts_dptbl[tspp->ts_umdpri].ts_globpri);
                ASSERT(curthread->t_pri >= 0 &&
                    curthread->t_pri <= ts_maxglobpri);
                tspp->ts_flags = flags & ~TSKPRI;

                if (DISP_MUST_SURRENDER(curthread))
                        cpu_surrender(curthread);
        }
        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
        TRACE_2(TR_FAC_DISP, TR_SLEEP,
            "sleep:tid %p old pri %d", t, old_pri);
}


/*
 * Return Values:
 *
 *      -1 if the thread is loaded or is not eligible to be swapped in.
 *
 *      effective priority of the specified thread based on swapout time
 *              and size of process (epri >= 0 , epri <= SHRT_MAX).
 */
/* ARGSUSED */
```

```
static pri_t
ts_swapin(kthread_t *t, int flags)
{
        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
        long            epri = -1;
        proc_t          *pp = ttoproc(t);

        ASSERT(THREAD_LOCK_HELD(t));

        /*
         * We know that pri_t is a short.
         * Be sure not to overrun its range.
         */
        if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
                time_t swapout_time;

                swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
                if (INHERITED(t) || (tspp->ts_flags & (TSKPRI | TSIASET)))
                        epri = (long)DISP_PRIO(t) + swapout_time;
                else {
                        /*
                         * Threads which have been out for a long time,
                         * have high user mode priority and are associated
                         * with a small address space are more deserving
                         */
                        epri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
                        ASSERT(epri >= 0 && epri <= ts_maxumdpri);
                        epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
                }
                /*
                 * Scale epri so SHRT_MAX/2 represents zero priority.
                 */
                epri += SHRT_MAX/2;
                if (epri < 0)
                        epri = 0;
                else if (epri > SHRT_MAX)
                        epri = SHRT_MAX;
        }
        return ((pri_t)epri);
}

/*
 * Return Values
 *      -1 if the thread isn't loaded or is not eligible to be swapped out.
 *
 *      effective priority of the specified thread based on if the swapper
 *              is in softswap or hardswap mode.
 *
 *              Softswap:  Return a low effective priority for threads
 *                         sleeping for more than maxslp secs.
 *
 *              Hardswap:  Return an effective priority such that threads
 *                         which have been in memory for a while and are
 *                         associated with a small address space are swapped
 *                         in before others.
 *
 *              (epri >= 0 , epri <= SHRT_MAX).
 */
time_t  ts_minrun = 2;          /* XXX - t_pri becomes 59 within 2 secs */
time_t  ts_minslp = 2;          /* min time on sleep queue for hardswap */

static pri_t
ts_swapout(kthread_t *t, int flags)
{
        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
        long            epri = -1;
        proc_t          *pp = ttoproc(t);
        time_t          swapin_time;

        ASSERT(THREAD_LOCK_HELD(t));

        if (INHERITED(t) || (tspp->ts_flags & (TSKPRI | TSIASET)) ||
            (t->t_proc_flag & TP_LWPEXIT) ||
            (t->t_state & (TS_ZOMB | TS_FREE | TS_STOPPED |
            TS_ONPROC | TS_WAIT)) ||
            !(t->t_schedflag & TS_LOAD) || !SWAP_OK(t))
                return (-1);

        ASSERT(t->t_state & (TS_SLEEP | TS_RUN));

        /*
         * We know that pri_t is a short.
         * Be sure not to overrun its range.
         */
        swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;
        if (flags == SOFTSWAP) {
                if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
                        epri = 0;
                } else {
                        return ((pri_t)epri);
                }
        } else {
                pri_t pri;

                if ((t->t_state == TS_SLEEP && swapin_time > ts_minslp) ||
                    (t->t_state == TS_RUN && swapin_time > ts_minrun)) {
                        pri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
                        ASSERT(pri >= 0 && pri <= ts_maxumdpri);
                        epri = swapin_time -
                            (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
                } else {
                        return ((pri_t)epri);
                }
        }

        /*
         * Scale epri so SHRT_MAX/2 represents zero priority.
         */
        epri += SHRT_MAX/2;
        if (epri < 0)
                epri = 0;
        else if (epri > SHRT_MAX)
                epri = SHRT_MAX;

        return ((pri_t)epri);
}

/*
 * Check for time slice expiration.  If time slice has expired
 * move thread to priority specified in tsdptbl for time slice expiration
 * and set runrun to cause preemption.
 */
static void
ts_tick(kthread_t *t)
{
        tsproc_t *tspp = (tsproc_t *)(t->t_cldata);
        klwp_t *lwp;
        boolean_t call_cpu_surrender = B_FALSE;
        pri_t   oldpri = t->t_pri;

        ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));
```

```
        thread_lock(t);

        /*
         * Keep track of thread's project CPU usage.  Note that projects
         * get charged even when threads are running in the kernel.
         */
        if (CPUCAPS_ON()) {
                call_cpu_surrender = cpucaps_charge(t, &tspp->ts_caps,
                    CPUCAPS_CHARGE_ENFORCE) && !(tspp->ts_flags & TSKPRI);
        }

        if ((tspp->ts_flags & TSKPRI) == 0) {
                if (--tspp->ts_timeleft <= 0) {
                        pri_t   new_pri;

                        /*
                         * If we're doing preemption control and trying to
                         * avoid preempting this thread, just note that
                         * the thread should yield soon and let it keep
                         * running (unless it's been a while).
                         */
                        if (t->t_schedctl && schedctl_get_nopreempt(t)) {
                                if (tspp->ts_timeleft > -SC_MAX_TICKS) {
                                        DTRACE_SCHED1(schedctl__nopreempt,
                                            kthread_t *, t);
                                        schedctl_set_yield(t, 1);
                                        thread_unlock_nopreempt(t);
                                        return;
                                }

                                TNF_PROBE_2(schedctl_failsafe,
                                    "schedctl TS ts_tick", /* CSTYLED */,
                                    tnf_pid, pid, ttoproc(t)->p_pid,
                                    tnf_lwpid, lwpid, t->t_tid);
                        }
                        tspp->ts_flags &= ~TSRESTORE;
                        tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_tqexp;
                        TS_NEWUMDPRI(tspp);
                        tspp->ts_dispwait = 0;
                        new_pri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
                        ASSERT(new_pri >= 0 && new_pri <= ts_maxglobpri);
                        /*
                         * When the priority of a thread is changed,
                         * it may be necessary to adjust its position
                         * on a sleep queue or dispatch queue.
                         * The function thread_change_pri accomplishes
                         * this.
                         */
                        if (thread_change_pri(t, new_pri, 0)) {
                                if ((t->t_schedflag & TS_LOAD) &&
                                    (lwp = t->t_lwp) &&
                                    lwp->lwp_state == LWP_USER)
                                        t->t_schedflag &= ~TS_DONT_SWAP;
                                tspp->ts_timeleft =
                                    ts_dptbl[tspp->ts_cpupri].ts_quantum;
                        } else {
                                call_cpu_surrender = B_TRUE;
                        }
                        TRACE_2(TR_FAC_DISP, TR_TICK,
                            "tick:tid %p old pri %d", t, oldpri);
                } else if (t->t_state == TS_ONPROC &&
                    t->t_pri < t->t_disp_queue->disp_maxrunpri) {
                        call_cpu_surrender = B_TRUE;
                }
        }
        if (call_cpu_surrender) {
                tspp->ts_flags |= TSBACKQ;
                cpu_surrender(t);
        }

        thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
}


/*
 * If thread is currently at a kernel mode priority (has slept)
 * we assign it the appropriate user mode priority and time quantum
 * here.  If we are lowering the thread's priority below that of
 * other runnable threads we will normally set runrun via cpu_surrender() to
 * cause preemption.
 */
static void
ts_trapret(kthread_t *t)
{
        tsproc_t        *tspp = (tsproc_t *)t->t_cldata;
        cpu_t           *cp = CPU;
        pri_t           old_pri = curthread->t_pri;

        ASSERT(THREAD_LOCK_HELD(t));
        ASSERT(t == curthread);
        ASSERT(cp->cpu_dispthread == t);
        ASSERT(t->t_state == TS_ONPROC);

        t->t_kpri_req = 0;
        if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
                tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
                TS_NEWUMDPRI(tspp);
                tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
                tspp->ts_dispwait = 0;

                /*
                 * If thread has blocked in the kernel (as opposed to
                 * being merely preempted), recompute the user mode priority.
                 */
                THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
                cp->cpu_dispatch_pri = DISP_PRIO(t);
                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
                tspp->ts_flags &= ~TSKPRI;

                if (DISP_MUST_SURRENDER(t))
                        cpu_surrender(t);
        } else if (tspp->ts_flags & TSKPRI) {
                /*
                 * If thread has blocked in the kernel (as opposed to
                 * being merely preempted), recompute the user mode priority.
                 */
                THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
                cp->cpu_dispatch_pri = DISP_PRIO(t);
                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
                tspp->ts_flags &= ~TSKPRI;

                if (DISP_MUST_SURRENDER(t))
                        cpu_surrender(t);
        }

        /*
         * Swapout lwp if the swapper is waiting for this thread to
         * reach a safe point.
         */
        if ((t->t_schedflag & TS_SWAPENQ) && !(tspp->ts_flags & TSIASET)) {
                thread_unlock(t);
```

```
                swapout_lwp(ttolwp(t));
                thread_lock(t);
        }

        TRACE_2(TR_FAC_DISP, TR_TRAPRET,
            "trapret:tid %p old pri %d", t, old_pri);
}
```
______***unchanged_portion_omitted_***

```
/*
 * Processes waking up go to the back of their queue.  We don't
 * need to assign a time quantum here because thread is still
 * at a kernel mode priority and the time slicing is not done
 * for threads running in the kernel after sleeping.  The proper
 * time quantum will be assigned by ts_trapret before the thread
 * returns to user mode.
 */
static void
ts_wakeup(kthread_t *t)
{
        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);

        ASSERT(THREAD_LOCK_HELD(t));

        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */

        if (tspp->ts_flags & TSKPRI) {
                tspp->ts_flags &= ~TSBACKQ;
                if (tspp->ts_flags & TSIASET)
                        setfrontdq(t);
                else
                        setbackdq(t);
        } else if (t->t_kpri_req) {
                /*
                 * Give thread a priority boost if we were asked.
                 */
                tspp->ts_flags |= TSKPRI;
                THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
                setbackdq(t);
                t->t_trapret = 1;       /* so that ts_trapret will run */
                aston(t);
        } else {
                if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
                        tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
                        TS_NEWUMDPRI(tspp);
                        tspp->ts_timeleft =
                            ts_dptbl[tspp->ts_cpupri].ts_quantum;
                        tspp->ts_dispwait = 0;
                        THREAD_CHANGE_PRI(t,
                            ts_dptbl[tspp->ts_umdpri].ts_globpri);
                        ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
                }

                tspp->ts_flags &= ~TSBACKQ;

                if (tspp->ts_flags & TSIA) {
                        if (tspp->ts_flags & TSIASET)
                                setfrontdq(t);
                        else
                                setbackdq(t);
                } else {
                        if (t->t_disp_time != ddi_get_lbolt())
                                setbackdq(t);
                        else
                                setfrontdq(t);
                }
        }
}
______unchanged_portion_omitted_
```

****
   67734 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/fs/nfs/nfs_srv.c
6583 remove whole-process swapping
****
*______unchanged_portion_omitted_*

```
static struct rfs_async_write_list *rfs_async_write_head = NULL;
static kmutex_t rfs_async_write_lock;
static int rfs_write_async = 1; /* enables write clustering if == 1 */

#define MAXCLIOVECS     42
#define RFSWRITE_INITVAL (enum nfsstat) -1

#ifdef DEBUG
static int rfs_write_hits = 0;
static int rfs_write_misses = 0;
#endif

/*
 * Write data to file.
 * Returns attributes of a file after writing some data to it.
 */
void
rfs_write(struct nfswriteargs *wa, struct nfsattrstat *ns,
    struct exportinfo *exi, struct svc_req *req, cred_t *cr, bool_t ro)
{
        int error;
        vnode_t *vp;
        rlim64_t rlimit;
        struct vattr va;
        struct uio uio;
        struct rfs_async_write_list *lp;
        struct rfs_async_write_list *nlp;
        struct rfs_async_write *rp;
        struct rfs_async_write *nrp;
        struct rfs_async_write *trp;
        struct rfs_async_write *lrp;
        int data_written;
        int iovcnt;
        mblk_t *m;
        struct iovec *iovp;
        struct iovec *niovp;
        struct iovec iov[MAXCLIOVECS];
        int count;
        int rcount;
        uint_t off;
        uint_t len;
        struct rfs_async_write nrpsp;
        struct rfs_async_write_list nlpsp;
        ushort_t t_flag;
        cred_t *savecred;
        int in_crit = 0;
        caller_context_t ct;

        if (!rfs_write_async) {
                rfs_write_sync(wa, ns, exi, req, cr, ro);
                return;
        }

        /*
         * Initialize status to RFSWRITE_INITVAL instead of 0, since value of 0
         * is considered an OK.
         */
        ns->ns_status = RFSWRITE_INITVAL;
```

```
        nrp = &nrpsp;
        nrp->wa = wa;
        nrp->ns = ns;
        nrp->req = req;
        nrp->cr = cr;
        nrp->ro = ro;
        nrp->thread = curthread;

        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

        /*
         * Look to see if there is already a cluster started
         * for this file.
         */
        mutex_enter(&rfs_async_write_lock);
        for (lp = rfs_async_write_head; lp != NULL; lp = lp->next) {
                if (bcmp(&wa->wa_fhandle, lp->fhp,
                    sizeof (fhandle_t)) == 0)
                        break;
        }

        /*
         * If lp is non-NULL, then there is already a cluster
         * started.  We need to place ourselves in the cluster
         * list in the right place as determined by starting
         * offset.  Conflicts with non-blocking mandatory locked
         * regions will be checked when the cluster is processed.
         */
        if (lp != NULL) {
                rp = lp->list;
                trp = NULL;
                while (rp != NULL && rp->wa->wa_offset < wa->wa_offset) {
                        trp = rp;
                        rp = rp->list;
                }
                nrp->list = rp;
                if (trp == NULL)
                        lp->list = nrp;
                else
                        trp->list = nrp;
                while (nrp->ns->ns_status == RFSWRITE_INITVAL)
                        cv_wait(&lp->cv, &rfs_async_write_lock);
                mutex_exit(&rfs_async_write_lock);

                return;
        }

        /*
         * No cluster started yet, start one and add ourselves
         * to the list of clusters.
         */
        nrp->list = NULL;

        nlp = &nlpsp;
        nlp->fhp = &wa->wa_fhandle;
        cv_init(&nlp->cv, NULL, CV_DEFAULT, NULL);
        nlp->list = nrp;
        nlp->next = NULL;

        if (rfs_async_write_head == NULL) {
                rfs_async_write_head = nlp;
        } else {
                lp = rfs_async_write_head;
                while (lp->next != NULL)
                        lp = lp->next;
                lp->next = nlp;
        }
        mutex_exit(&rfs_async_write_lock);

        /*
         * Convert the file handle common to all of the requests
         * in this cluster to a vnode.
         */
        vp = nfs_fhtovp(&wa->wa_fhandle, exi);
        if (vp == NULL) {
                mutex_enter(&rfs_async_write_lock);
                if (rfs_async_write_head == nlp)
                        rfs_async_write_head = nlp->next;
                else {
                        lp = rfs_async_write_head;
                        while (lp->next != nlp)
                                lp = lp->next;
                        lp->next = nlp->next;
                }
                t_flag = curthread->t_flag & T_WOULDBLOCK;
                for (rp = nlp->list; rp != NULL; rp = rp->list) {
                        rp->ns->ns_status = NFSERR_STALE;
                        rp->thread->t_flag |= t_flag;
                }
                cv_broadcast(&nlp->cv);
                mutex_exit(&rfs_async_write_lock);

                return;
        }

        /*
         * Can only write regular files.  Attempts to write any
         * other file types fail with EISDIR.
         */
        if (vp->v_type != VREG) {
                VN_RELE(vp);
                mutex_enter(&rfs_async_write_lock);
                if (rfs_async_write_head == nlp)
                        rfs_async_write_head = nlp->next;
                else {
                        lp = rfs_async_write_head;
                        while (lp->next != nlp)
                                lp = lp->next;
                        lp->next = nlp->next;
                }
                t_flag = curthread->t_flag & T_WOULDBLOCK;
                for (rp = nlp->list; rp != NULL; rp = rp->list) {
                        rp->ns->ns_status = NFSERR_ISDIR;
                        rp->thread->t_flag |= t_flag;
                }
                cv_broadcast(&nlp->cv);
                mutex_exit(&rfs_async_write_lock);

                return;
        }

        /*
         * Enter the critical region before calling VOP_RWLOCK, to avoid a
         * deadlock with ufs.
         */
        if (nbl_need_check(vp)) {
                nbl_start_crit(vp, RW_READER);
                in_crit = 1;
        }

        ct.cc_sysid = 0;
        ct.cc_pid = 0;
```

```
        ct.cc_caller_id = nfs2_srv_caller_id;
        ct.cc_flags = CC_DONTBLOCK;

        /*
         * Lock the file for writing.  This operation provides
         * the delay which allows clusters to grow.
         */
        error = VOP_RWLOCK(vp, V_WRITELOCK_TRUE, &ct);

        /* check if a monitor detected a delegation conflict */
        if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK)) {
                if (in_crit)
                        nbl_end_crit(vp);
                VN_RELE(vp);
                /* mark as wouldblock so response is dropped */
                curthread->t_flag |= T_WOULDBLOCK;
                mutex_enter(&rfs_async_write_lock);
                if (rfs_async_write_head == nlp)
                        rfs_async_write_head = nlp->next;
                else {
                        lp = rfs_async_write_head;
                        while (lp->next != nlp)
                                lp = lp->next;
                        lp->next = nlp->next;
                }
                for (rp = nlp->list; rp != NULL; rp = rp->list) {
                        if (rp->ns->ns_status == RFSWRITE_INITVAL) {
                                rp->ns->ns_status = puterrno(error);
                                rp->thread->t_flag |= T_WOULDBLOCK;
                        }
                }
                cv_broadcast(&nlp->cv);
                mutex_exit(&rfs_async_write_lock);

                return;
        }

        /*
         * Disconnect this cluster from the list of clusters.
         * The cluster that is being dealt with must be fixed
         * in size after this point, so there is no reason
         * to leave it on the list so that new requests can
         * find it.
         *
         * The algorithm is that the first write request will
         * create a cluster, convert the file handle to a
         * vnode pointer, and then lock the file for writing.
         * This request is not likely to be clustered with
         * any others.  However, the next request will create
         * a new cluster and be blocked in VOP_RWLOCK while
         * the first request is being processed.  This delay
         * will allow more requests to be clustered in this
         * second cluster.
         */
        mutex_enter(&rfs_async_write_lock);
        if (rfs_async_write_head == nlp)
                rfs_async_write_head = nlp->next;
        else {
                lp = rfs_async_write_head;
                while (lp->next != nlp)
                        lp = lp->next;
                lp->next = nlp->next;
        }
        mutex_exit(&rfs_async_write_lock);

        /*
```

```
         * Step through the list of requests in this cluster.
         * We need to check permissions to make sure that all
         * of the requests have sufficient permission to write
         * the file.  A cluster can be composed of requests
         * from different clients and different users on each
         * client.
         *
         * As a side effect, we also calculate the size of the
         * byte range that this cluster encompasses.
         */
        rp = nlp->list;
        off = rp->wa->wa_offset;
        len = (uint_t)0;
        do {
                if (rdonly(rp->ro, vp)) {
                        rp->ns->ns_status = NFSERR_ROFS;
                        t_flag = curthread->t_flag & T_WOULDBLOCK;
                        rp->thread->t_flag |= t_flag;
                        continue;
                }

                va.va_mask = AT_UID|AT_MODE;

                error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

                if (!error) {
                        if (crgetuid(rp->cr) != va.va_uid) {
                                /*
                                 * This is a kludge to allow writes of files
                                 * created with read only permission.  The
                                 * owner of the file is always allowed to
                                 * write it.
                                 */
                                error = VOP_ACCESS(vp, VWRITE, 0, rp->cr, &ct);
                        }
                        if (!error && MANDLOCK(vp, va.va_mode))
                                error = EACCES;
                }

                /*
                 * Check for a conflict with a nbmand-locked region.
                 */
                if (in_crit && nbl_conflict(vp, NBL_WRITE, rp->wa->wa_offset,
                    rp->wa->wa_count, 0, NULL)) {
                        error = EACCES;
                }

                if (error) {
                        rp->ns->ns_status = puterrno(error);
                        t_flag = curthread->t_flag & T_WOULDBLOCK;
                        rp->thread->t_flag |= t_flag;
                        continue;
                }
                if (len < rp->wa->wa_offset + rp->wa->wa_count - off)
                        len = rp->wa->wa_offset + rp->wa->wa_count - off;
        } while ((rp = rp->list) != NULL);

        /*
         * Step through the cluster attempting to gather as many
         * requests which are contiguous as possible.  These
         * contiguous requests are handled via one call to VOP_WRITE
         * instead of different calls to VOP_WRITE.  We also keep
         * track of the fact that any data was written.
         */
        rp = nlp->list;
        data_written = 0;
```

```
        do {
                /*
                 * Skip any requests which are already marked as having an
                 * error.
                 */
                if (rp->ns->ns_status != RFSWRITE_INITVAL) {
                        rp = rp->list;
                        continue;
                }

                /*
                 * Count the number of iovec's which are required
                 * to handle this set of requests.  One iovec is
                 * needed for each data buffer, whether addressed
                 * by wa_data or by the b_rptr pointers in the
                 * mblk chains.
                 */
                iovcnt = 0;
                lrp = rp;
                for (;;) {
                        if (lrp->wa->wa_data || lrp->wa->wa_rlist)
                                iovcnt++;
                        else {
                                m = lrp->wa->wa_mblk;
                                while (m != NULL) {
                                        iovcnt++;
                                        m = m->b_cont;
                                }
                        }
                        if (lrp->list == NULL ||
                            lrp->list->ns->ns_status != RFSWRITE_INITVAL ||
                            lrp->wa->wa_offset + lrp->wa->wa_count !=
                            lrp->list->wa->wa_offset) {
                                lrp = lrp->list;
                                break;
                        }
                        lrp = lrp->list;
                }

                if (iovcnt <= MAXCLIOVECS) {
#ifdef DEBUG
                        rfs_write_hits++;
#endif
                        niovp = iov;
                } else {
#ifdef DEBUG
                        rfs_write_misses++;
#endif
                        niovp = kmem_alloc(sizeof (*niovp) * iovcnt, KM_SLEEP);
                }
                /*
                 * Put together the scatter/gather iovecs.
                 */
                iovp = niovp;
                trp = rp;
                count = 0;
                do {
                        if (trp->wa->wa_data || trp->wa->wa_rlist) {
                                if (trp->wa->wa_rlist) {
                                        iovp->iov_base =
                                            (char *)((trp->wa->wa_rlist)->
                                            u.c_daddr3);
                                        iovp->iov_len = trp->wa->wa_count;
                                } else  {
                                        iovp->iov_base = trp->wa->wa_data;
                                        iovp->iov_len = trp->wa->wa_count;
                                }
                                iovp++;
                        } else {
                                m = trp->wa->wa_mblk;
                                rcount = trp->wa->wa_count;
                                while (m != NULL) {
                                        iovp->iov_base = (caddr_t)m->b_rptr;
                                        iovp->iov_len = (m->b_wptr - m->b_rptr);
                                        rcount -= iovp->iov_len;
                                        if (rcount < 0)
                                                iovp->iov_len += rcount;
                                        iovp++;
                                        if (rcount <= 0)
                                                break;
                                        m = m->b_cont;
                                }
                        }
                        count += trp->wa->wa_count;
                        trp = trp->list;
                } while (trp != lrp);

                uio.uio_iov = niovp;
                uio.uio_iovcnt = iovcnt;
                uio.uio_segflg = UIO_SYSSPACE;
                uio.uio_extflg = UIO_COPY_DEFAULT;
                uio.uio_loffset = (offset_t)rp->wa->wa_offset;
                uio.uio_resid = count;
                /*
                 * The limit is checked on the client. We
                 * should allow any size writes here.
                 */
                uio.uio_llimit = curproc->p_fsz_ctl;
                rlimit = uio.uio_llimit - rp->wa->wa_offset;
                if (rlimit < (rlim64_t)uio.uio_resid)
                        uio.uio_resid = (uint_t)rlimit;

                /*
                 * For now we assume no append mode.
                 */

                /*
                 * We’re changing creds because VM may fault
                 * and we need the cred of the current
                 * thread to be used if quota * checking is
                 * enabled.
                 */
                savecred = curthread->t_cred;
                curthread->t_cred = cr;
                error = VOP_WRITE(vp, &uio, 0, rp->cr, &ct);
                curthread->t_cred = savecred;

                /* check if a monitor detected a delegation conflict */
                if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
                        /* mark as wouldblock so response is dropped */
                        curthread->t_flag |= T_WOULDBLOCK;

                if (niovp != iov)
                        kmem_free(niovp, sizeof (*niovp) * iovcnt);

                if (!error) {
                        data_written = 1;
                        /*
                         * Get attributes again so we send the latest mod
                         * time to the client side for his cache.
                         */
                        va.va_mask = AT_ALL;    /* now we want everything */
```

```
                                error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

                                if (!error)
                                        acl_perm(vp, exi, &va, rp->cr);
                        }

                        /*
                         * Fill in the status responses for each request
                         * which was just handled.  Also, copy the latest
                         * attributes in to the attribute responses if
                         * appropriate.
                         */
                        t_flag = curthread->t_flag & T_WOULDBLOCK;
                        do {
                                rp->thread->t_flag |= t_flag;
                                /* check for overflows */
                                if (!error) {
                                        error  = vattr_to_nattr(&va, &rp->ns->ns_attr);
                                }
                                rp->ns->ns_status = puterrno(error);
                                rp = rp->list;
                        } while (rp != lrp);
                } while (rp != NULL);

                /*
                 * If any data was written at all, then we need to flush
                 * the data and metadata to stable storage.
                 */
                if (data_written) {
                        error = VOP_PUTPAGE(vp, (u_offset_t)off, len, 0, cr, &ct);

                        if (!error) {
                                error = VOP_FSYNC(vp, FNODSYNC, cr, &ct);
                        }
                }

                VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, &ct);

                if (in_crit)
                        nbl_end_crit(vp);
                VN_RELE(vp);

                t_flag = curthread->t_flag & T_WOULDBLOCK;
                mutex_enter(&rfs_async_write_lock);
                for (rp = nlp->list; rp != NULL; rp = rp->list) {
                        if (rp->ns->ns_status == RFSWRITE_INITVAL) {
                                rp->ns->ns_status = puterrno(error);
                                rp->thread->t_flag |= t_flag;
                        }
                }
                cv_broadcast(&nlp->cv);
                mutex_exit(&rfs_async_write_lock);

        }
______unchanged_portion_omitted_
```

```
************************************************************
   73921 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/os/clock.c
6583 remove whole-process swapping
************************************************************
______unchanged_portion_omitted_

/*
 * test hook for tod broken detection in tod_validate
 */
int tod_unit_test = 0;
time_t tod_test_injector;

#define CLOCK_ADJ_HIST_SIZE     4

static int      adj_hist_entry;

int64_t clock_adj_hist[CLOCK_ADJ_HIST_SIZE];

static void calcloadavg(int, uint64_t *);
static int genloadavg(struct loadavg_s *);
static void loadavg_update();

void (*cmm_clock_callout)() = NULL;
void (*cpucaps_clock_callout)() = NULL;

extern clock_t clock_tick_proc_max;

static int64_t deadman_counter = 0;

static void
clock(void)
{
        kthread_t       *t;
        uint_t  nrunnable;
        uint_t  w_io;
        cpu_t   *cp;
        cpupart_t *cpupart;
        extern  void    set_freemem();
        void    (*funcp)();
        int32_t ltemp;
        int64_t lltemp;
        int s;
        int do_lgrp_load;
        int i;
        clock_t now = LBOLT_NO_ACCOUNT; /* current tick */

        if (panicstr)
                return;

        /*
         * Make sure that 'freemem' do not drift too far from the truth
         */
        set_freemem();


        /*
         * Before the section which is repeated is executed, we do
         * the time delta processing which occurs every clock tick
         *
         * There is additional processing which happens every time
         * the nanosecond counter rolls over which is described
         * below - see the section which begins with : if (one_sec)
         *
         * This section marks the beginning of the precision-kernel
         * code fragment.
```

```
         *
         * First, compute the phase adjustment. If the low-order bits
         * (time_phase) of the update overflow, bump the higher order
         * bits (time_update).
         */
        time_phase += time_adj;
        if (time_phase <= -FINEUSEC) {
                ltemp = -time_phase / SCALE_PHASE;
                time_phase += ltemp * SCALE_PHASE;
                s = hr_clock_lock();
                timedelta -= ltemp * (NANOSEC/MICROSEC);
                hr_clock_unlock(s);
        } else if (time_phase >= FINEUSEC) {
                ltemp = time_phase / SCALE_PHASE;
                time_phase -= ltemp * SCALE_PHASE;
                s = hr_clock_lock();
                timedelta += ltemp * (NANOSEC/MICROSEC);
                hr_clock_unlock(s);
        }

        /*
         * End of precision-kernel code fragment which is processed
         * every timer interrupt.
         *
         * Continue with the interrupt processing as scheduled.
         */
        /*
         * Count the number of runnable threads and the number waiting
         * for some form of I/O to complete -- gets added to
         * sysinfo.waiting.  To know the state of the system, must add
         * wait counts from all CPUs.  Also add up the per-partition
         * statistics.
         */
        w_io = 0;
        nrunnable = 0;

        /*
         * keep track of when to update lgrp/part loads
         */

        do_lgrp_load = 0;
        if (lgrp_ticks++ >= hz / 10) {
                lgrp_ticks = 0;
                do_lgrp_load = 1;
        }

        if (one_sec) {
                loadavg_update();
                deadman_counter++;
        }

        /*
         * First count the threads waiting on kpreempt queues in each
         * CPU partition.
         */

        cpupart = cp_list_head;
        do {
                uint_t cpupart_nrunnable = cpupart->cp_kp_queue.disp_nrunnable;

                cpupart->cp_updates++;
                nrunnable += cpupart_nrunnable;
                cpupart->cp_nrunnable_cum += cpupart_nrunnable;
                if (one_sec) {
                        cpupart->cp_nrunning = 0;
                        cpupart->cp_nrunnable = cpupart_nrunnable;
```

```
		}
	} while ((cpupart = cpupart->cp_next) != cp_list_head);


	/* Now count the per-CPU statistics. */
	cp = cpu_list;
	do {
		uint_t cpu_nrunnable = cp->cpu_disp->disp_nrunnable;

		nrunnable += cpu_nrunnable;
		cpupart = cp->cpu_part;
		cpupart->cp_nrunnable_cum += cpu_nrunnable;
		if (one_sec) {
			cpupart->cp_nrunnable += cpu_nrunnable;
			/*
			 * Update user, system, and idle cpu times.
			 */
			cpupart->cp_nrunning++;
			/*
			 * w_io is used to update sysinfo.waiting during
			 * one_second processing below.  Only gather w_io
			 * information when we walk the list of cpus if we're
			 * going to perform one_second processing.
			 */
			w_io += CPU_STATS(cp, sys.iowait);
		}

		if (one_sec && (cp->cpu_flags & CPU_EXISTS)) {
			int i, load, change;
			hrtime_t intracct, intrused;
			const hrtime_t maxnsec = 1000000000;
			const int precision = 100;

			/*
			 * Estimate interrupt load on this cpu each second.
			 * Computes cpu_intrload as %utilization (0-99).
			 */

			/* add up interrupt time from all micro states */
			for (intracct = 0, i = 0; i < NCMSTATES; i++)
				intracct += cp->cpu_intracct[i];
			scalehrtime(&intracct);

			/* compute nsec used in the past second */
			intrused = intracct - cp->cpu_intrlast;
			cp->cpu_intrlast = intracct;

			/* limit the value for safety (and the first pass) */
			if (intrused >= maxnsec)
				intrused = maxnsec - 1;

			/* calculate %time in interrupt */
			load = (precision * intrused) / maxnsec;
			ASSERT(load >= 0 && load < precision);
			change = cp->cpu_intrload - load;

			/* jump to new max, or decay the old max */
			if (change < 0)
				cp->cpu_intrload = load;
			else if (change > 0)
				cp->cpu_intrload -= (change + 3) / 4;

			DTRACE_PROBE3(cpu_intrload,
			    cpu_t *, cp,
			    hrtime_t, intracct,
			    hrtime_t, intrused);
```

```
		}

		if (do_lgrp_load &&
		    (cp->cpu_flags & CPU_EXISTS)) {
			/*
			 * When updating the lgroup's load average,
			 * account for the thread running on the CPU.
			 * If the CPU is the current one, then we need
			 * to account for the underlying thread which
			 * got the clock interrupt not the thread that is
			 * handling the interrupt and caculating the load
			 * average
			 */
			t = cp->cpu_thread;
			if (CPU == cp)
				t = t->t_intr;

			/*
			 * Account for the load average for this thread if
			 * it isn't the idle thread or it is on the interrupt
			 * stack and not the current CPU handling the clock
			 * interrupt
			 */
			if ((t && t != cp->cpu_idle_thread) || (CPU != cp &&
			    CPU_ON_INTR(cp))) {
				if (t->t_lpl == cp->cpu_lpl) {
					/* local thread */
					cpu_nrunnable++;
				} else {
					/*
					 * This is a remote thread, charge it
					 * against its home lgroup.  Note that
					 * we notice that a thread is remote
					 * only if it's currently executing.
					 * This is a reasonable approximation,
					 * since queued remote threads are rare.
					 * Note also that if we didn't charge
					 * it to its home lgroup, remote
					 * execution would often make a system
					 * appear balanced even though it was
					 * not, and thread placement/migration
					 * would often not be done correctly.
					 */
					lgrp_loadavg(t->t_lpl,
					    LGRP_LOADAVG_IN_THREAD_MAX, 0);
				}
			}
			lgrp_loadavg(cp->cpu_lpl,
			    cpu_nrunnable * LGRP_LOADAVG_IN_THREAD_MAX, 1);
		}
	} while ((cp = cp->cpu_next) != cpu_list);

	clock_tick_schedule(one_sec);

	/*
	 * Check for a callout that needs be called from the clock
	 * thread to support the membership protocol in a clustered
	 * system.  Copy the function pointer so that we can reset
	 * this to NULL if needed.
	 */
	if ((funcp = cmm_clock_callout) != NULL)
		(*funcp)();

	if ((funcp = cpucaps_clock_callout) != NULL)
		(*funcp)();
	/*
	 * Wakeup the cageout thread waiters once per second.
	 */
	if (one_sec)
		kcage_tick();

	if (one_sec) {

		int drift, absdrift;
		timestruc_t tod;
		int s;

		/*
		 * Beginning of precision-kernel code fragment executed
		 * every second.
		 *
		 * On rollover of the second the phase adjustment to be
		 * used for the next second is calculated.  Also, the
		 * maximum error is increased by the tolerance.  If the
		 * PPS frequency discipline code is present, the phase is
		 * increased to compensate for the CPU clock oscillator
		 * frequency error.
		 *
		 * On a 32-bit machine and given parameters in the timex.h
		 * header file, the maximum phase adjustment is +-512 ms
		 * and maximum frequency offset is (a tad less than)
		 * +-512 ppm. On a 64-bit machine, you shouldn’t need to ask.
		 */
		time_maxerror += time_tolerance / SCALE_USEC;

		/*
		 * Leap second processing. If in leap-insert state at
		 * the end of the day, the system clock is set back one
		 * second; if in leap-delete state, the system clock is
		 * set ahead one second. The microtime() routine or
		 * external clock driver will insure that reported time
		 * is always monotonic. The ugly divides should be
		 * replaced.
		 */
		switch (time_state) {

		case TIME_OK:
			if (time_status & STA_INS)
				time_state = TIME_INS;
			else if (time_status & STA_DEL)
				time_state = TIME_DEL;
			break;

		case TIME_INS:
			if (hrestime.tv_sec % 86400 == 0) {
				s = hr_clock_lock();
				hrestime.tv_sec--;
				hr_clock_unlock(s);
				time_state = TIME_OOP;
			}
			break;

		case TIME_DEL:
			if ((hrestime.tv_sec + 1) % 86400 == 0) {
				s = hr_clock_lock();
				hrestime.tv_sec++;
				hr_clock_unlock(s);
				time_state = TIME_WAIT;
			}
			break;
```

```
		case TIME_OOP:
			time_state = TIME_WAIT;
			break;

		case TIME_WAIT:
			if (!(time_status & (STA_INS | STA_DEL)))
				time_state = TIME_OK;
		default:
			break;
		}

		/*
		 * Compute the phase adjustment for the next second. In
		 * PLL mode, the offset is reduced by a fixed factor
		 * times the time constant. In FLL mode the offset is
		 * used directly. In either mode, the maximum phase
		 * adjustment for each second is clamped so as to spread
		 * the adjustment over not more than the number of
		 * seconds between updates.
		 */
		if (time_offset == 0)
			time_adj = 0;
		else if (time_offset < 0) {
			lltemp = -time_offset;
			if (!(time_status & STA_FLL)) {
				if ((1 << time_constant) >= SCALE_KG)
					lltemp *= (1 << time_constant) /
					    SCALE_KG;
				else
					lltemp = (lltemp / SCALE_KG) >>
					    time_constant;
			}
			if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
				lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
			time_offset += lltemp;
			time_adj = -(lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
		} else {
			lltemp = time_offset;
			if (!(time_status & STA_FLL)) {
				if ((1 << time_constant) >= SCALE_KG)
					lltemp *= (1 << time_constant) /
					    SCALE_KG;
				else
					lltemp = (lltemp / SCALE_KG) >>
					    time_constant;
			}
			if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
				lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
			time_offset -= lltemp;
			time_adj = (lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
		}

		/*
		 * Compute the frequency estimate and additional phase
		 * adjustment due to frequency error for the next
		 * second. When the PPS signal is engaged, gnaw on the
		 * watchdog counter and update the frequency computed by
		 * the pll and the PPS signal.
		 */
		pps_valid++;
		if (pps_valid == PPS_VALID) {
			pps_jitter = MAXTIME;
			pps_stabil = MAXFREQ;
			time_status &= ~(STA_PPSSIGNAL | STA_PPSJITTER |
			    STA_PPSWANDER | STA_PPSERROR);
		}
		lltemp = time_freq + pps_freq;

		if (lltemp)
			time_adj += (lltemp * SCALE_PHASE) / (SCALE_USEC * hz);

		/*
		 * End of precision kernel-code fragment
		 *
		 * The section below should be modified if we are planning
		 * to use NTP for synchronization.
		 *
		 * Note: the clock synchronization code now assumes
		 * the following:
		 *   - if dosynctodr is 1, then compute the drift between
		 *	the tod chip and software time and adjust one or
		 *	the other depending on the circumstances
		 *
		 *   - if dosynctodr is 0, then the tod chip is independent
		 *	of the software clock and should not be adjusted,
		 *	but allowed to free run.  this allows NTP to sync.
		 *	hrestime without any interference from the tod chip.
		 */

		tod_validate_deferred = B_FALSE;
		mutex_enter(&tod_lock);
		tod = tod_get();
		drift = tod.tv_sec - hrestime.tv_sec;
		absdrift = (drift >= 0) ? drift : -drift;
		if (tod_needsync || absdrift > 1) {
			int s;
			if (absdrift > 2) {
				if (!tod_broken && tod_faulted == TOD_NOFAULT) {
					s = hr_clock_lock();
					hrestime = tod;
					membar_enter(); /* hrestime visible */
					timedelta = 0;
					timechanged++;
					tod_needsync = 0;
					hr_clock_unlock(s);
					callout_hrestime();

				}
			} else {
				if (tod_needsync || !dosynctodr) {
					gethrestime(&tod);
					tod_set(tod);
					s = hr_clock_lock();
					if (timedelta == 0)
						tod_needsync = 0;
					hr_clock_unlock(s);
				} else {
					/*
					 * If the drift is 2 seconds on the
					 * money, then the TOD is adjusting
					 * the clock;  record that.
					 */
					clock_adj_hist[adj_hist_entry++ %
					    CLOCK_ADJ_HIST_SIZE] = now;
					s = hr_clock_lock();
					timedelta = (int64_t)drift*NANOSEC;
					hr_clock_unlock(s);
				}
			}
		}
		one_sec = 0;
		time = gethrestime_sec();  /* for crusty old kmem readers */
```

```
		mutex_exit(&tod_lock);

		/*
		 * Some drivers still depend on this... XXX
		 */
		cv_broadcast(&lbolt_cv);

		vminfo.freemem += freemem;
		{
			pgcnt_t maxswap, resv, free;
			pgcnt_t avail =
			    MAX((spgcnt_t)(availrmem - swapfs_minfree), 0);

			maxswap = k_anoninfo.ani_mem_resv +
			    k_anoninfo.ani_max +avail;
			/* Update ani_free */
			set_anoninfo();
			free = k_anoninfo.ani_free + avail;
			resv = k_anoninfo.ani_phys_resv +
			    k_anoninfo.ani_mem_resv;

			vminfo.swap_resv += resv;
			/* number of reserved and allocated pages */
#ifdef	DEBUG
			if (maxswap < free)
				cmn_err(CE_WARN, "clock: maxswap < free");
			if (maxswap < resv)
				cmn_err(CE_WARN, "clock: maxswap < resv");
#endif
			vminfo.swap_alloc += maxswap - free;
			vminfo.swap_avail += maxswap - resv;
			vminfo.swap_free += free;
		}
		vminfo.updates++;
		if (nrunnable) {
			sysinfo.runque += nrunnable;
			sysinfo.runocc++;
		}
		if (nswapped) {
			sysinfo.swpque += nswapped;
			sysinfo.swpocc++;
		}
		sysinfo.waiting += w_io;
		sysinfo.updates++;

		/*
		 * Wake up fsflush to write out DELWRI
		 * buffers, dirty pages and other cached
		 * administrative data, e.g. inodes.
		 */
		if (--fsflushcnt <= 0) {
			fsflushcnt = tune.t_fsflushr;
			cv_signal(&fsflush_cv);
		}

		vmmeter();
		calcloadavg(genloadavg(&loadavg), hp_avenrun);
		for (i = 0; i < 3; i++)
			/*
			 * At the moment avenrun[] can only hold 31
			 * bits of load average as it is a signed
			 * int in the API. We need to ensure that
			 * hp_avenrun[i] >> (16 - FSHIFT) will not be
			 * too large. If it is, we put the largest value
			 * that we can use into avenrun[i]. This is
			 * kludgey, but about all we can do until we
			 * avenrun[] is declared as an array of uint64[]
			 */
			if (hp_avenrun[i] < ((uint64_t)1<<(31+16-FSHIFT)))
				avenrun[i] = (int32_t)(hp_avenrun[i] >>
				    (16 - FSHIFT));
			else
				avenrun[i] = 0x7fffffff;

		cpupart = cp_list_head;
		do {
			calcloadavg(genloadavg(&cpupart->cp_loadavg),
			    cpupart->cp_hp_avenrun);
		} while ((cpupart = cpupart->cp_next) != cp_list_head);

		/*
		 * Wake up the swapper thread if necessary.
		 */
		if (runin ||
		    (runout && (avefree < desfree || wake_sched_sec))) {
			t = &t0;
			thread_lock(t);
			if (t->t_state == TS_STOPPED) {
				runin = runout = 0;
				wake_sched_sec = 0;
				t->t_whystop = 0;
				t->t_whatstop = 0;
				t->t_schedflag &= ~TS_ALLSTART;
				THREAD_TRANSITION(t);
				setfrontdq(t);
			}
			thread_unlock(t);
		}
	}

	/*
	 * Wake up the swapper if any high priority swapped-out threads
	 * became runable during the last tick.
	 */
	if (wake_sched) {
		t = &t0;
		thread_lock(t);
		if (t->t_state == TS_STOPPED) {
			runin = runout = 0;
			wake_sched = 0;
			t->t_whystop = 0;
			t->t_whatstop = 0;
			t->t_schedflag &= ~TS_ALLSTART;
			THREAD_TRANSITION(t);
			setfrontdq(t);
		}
		thread_unlock(t);
	}
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   21374 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/condvar.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#define cv_block_sig(t, cvp)    \
        { (t)->t_flag |= T_WAKEABLE; cv_block(cvp); }

/*
 * Block on the indicated condition variable and release the
 * associated kmutex while blocked.
 */
void
cv_wait(kcondvar_t *cvp, kmutex_t *mp)
{
        if (panicstr)
                return;
        ASSERT(!quiesce_active);

        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
        thread_lock(curthread);                 /* lock the thread */
        cv_block((condvar_impl_t *)cvp);
        thread_unlock_nopreempt(curthread);     /* unlock the waiters field */
        mutex_exit(mp);
        swtch();
        mutex_enter(mp);
}
______unchanged_portion_omitted_

int
cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp)
{
        kthread_t *t = curthread;
        proc_t *p = ttoproc(t);
        klwp_t *lwp = ttolwp(t);
        int cancel_pending;
        int rval = 1;
        int signalled = 0;

        if (panicstr)
                return (rval);
        ASSERT(!quiesce_active);

        /*
         * Threads in system processes don't process signals.  This is
         * true both for standard threads of system processes and for
         * interrupt threads which have borrowed their pinned thread's LWP.
         */
        if (lwp == NULL || (p->p_flag & SSYS)) {
                cv_wait(cvp, mp);
                return (rval);
        }
        ASSERT(t->t_intr == NULL);

        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
        cancel_pending = schedctl_cancel_pending();
        lwp->lwp_asleep = 1;
        lwp->lwp_sysabort = 0;
        thread_lock(t);
        cv_block_sig(t, (condvar_impl_t *)cvp);
        thread_unlock_nopreempt(t);
        mutex_exit(mp);
        if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
                setrun(t);
```

```
        /* ASSERT(no locks are held) */
        swtch();
        signalled = (t->t_schedflag & TS_SIGNALLED);
        t->t_flag &= ~T_WAKEABLE;
        mutex_enter(mp);
        if (ISSIG_PENDING(t, lwp, p)) {
                mutex_exit(mp);
                if (issig(FORREAL))
                        rval = 0;
                mutex_enter(mp);
        }
        if (lwp->lwp_sysabort || MUSTRETURN(p, t))
                rval = 0;
        if (rval != 0 && cancel_pending) {
                schedctl_cancel_eintr();
                rval = 0;
        }
        lwp->lwp_asleep = 0;
        lwp->lwp_sysabort = 0;
        if (rval == 0 && signalled)     /* avoid consuming the cv_signal() */
                cv_signal(cvp);
        return (rval);
}
______unchanged_portion_omitted_

/*
 * Like cv_wait_sig_swap but allows the caller to indicate (with a
 * non-NULL sigret) that they will take care of signalling the cv
 * after wakeup, if necessary.  This is a vile hack that should only
 * be used when no other option is available; almost all callers
 * should just use cv_wait_sig_swap (which takes care of the cv_signal
 * stuff automatically) instead.
 */
int
cv_wait_sig_swap_core(kcondvar_t *cvp, kmutex_t *mp, int *sigret)
{
        kthread_t *t = curthread;
        proc_t *p = ttoproc(t);
        klwp_t *lwp = ttolwp(t);
        int cancel_pending;
        int rval = 1;
        int signalled = 0;

        if (panicstr)
                return (rval);

        /*
         * Threads in system processes don't process signals.  This is
         * true both for standard threads of system processes and for
         * interrupt threads which have borrowed their pinned thread's LWP.
         */
        if (lwp == NULL || (p->p_flag & SSYS)) {
                cv_wait(cvp, mp);
                return (rval);
        }
        ASSERT(t->t_intr == NULL);

        cancel_pending = schedctl_cancel_pending();
        lwp->lwp_asleep = 1;
        lwp->lwp_sysabort = 0;
        thread_lock(t);
        t->t_kpri_req = 0;      /* don't need kernel priority */
        cv_block_sig(t, (condvar_impl_t *)cvp);
        /* I can be swapped now */
        curthread->t_schedflag &= ~TS_DONT_SWAP;
        thread_unlock_nopreempt(t);
        mutex_exit(mp);
        if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
                setrun(t);
        /* ASSERT(no locks are held) */
        swtch();
        signalled = (t->t_schedflag & TS_SIGNALLED);
        t->t_flag &= ~T_WAKEABLE;
        /* TS_DONT_SWAP set by disp() */
        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
        mutex_enter(mp);
        if (ISSIG_PENDING(t, lwp, p)) {
                mutex_exit(mp);
                if (issig(FORREAL))
                        rval = 0;
                mutex_enter(mp);
        }
        if (lwp->lwp_sysabort || MUSTRETURN(p, t))
                rval = 0;
        if (rval != 0 && cancel_pending) {
                schedctl_cancel_eintr();
                rval = 0;
        }
        lwp->lwp_asleep = 0;
        lwp->lwp_sysabort = 0;
        if (rval == 0) {
                if (sigret != NULL)
                        *sigret = signalled;    /* just tell the caller */
                else if (signalled)
                        cv_signal(cvp); /* avoid consuming the cv_signal() */
        }
        return (rval);
}
```
______***unchanged_portion_omitted***_

```
************************************************************
   93874 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/cpu.c
6583 remove whole-process swapping
************************************************************
______unchanged_portion_omitted_

static struct cpu_vm_stats_ks_data {
        kstat_named_t pgrec;
        kstat_named_t pgfrec;
        kstat_named_t pgin;
        kstat_named_t pgpgin;
        kstat_named_t pgout;
        kstat_named_t pgpgout;
        kstat_named_t swapin;
        kstat_named_t pgswapin;
        kstat_named_t swapout;
        kstat_named_t pgswapout;
        kstat_named_t zfod;
        kstat_named_t dfree;
        kstat_named_t scan;
        kstat_named_t rev;
        kstat_named_t hat_fault;
        kstat_named_t as_fault;
        kstat_named_t maj_fault;
        kstat_named_t cow_fault;
        kstat_named_t prot_fault;
        kstat_named_t softlock;
        kstat_named_t kernel_asflt;
        kstat_named_t pgrrun;
        kstat_named_t execpgin;
        kstat_named_t execpgout;
        kstat_named_t execfree;
        kstat_named_t anonpgin;
        kstat_named_t anonpgout;
        kstat_named_t anonfree;
        kstat_named_t fspgin;
        kstat_named_t fspgout;
        kstat_named_t fsfree;
} cpu_vm_stats_ks_data_template = {
        { "pgrec",              KSTAT_DATA_UINT64 },
        { "pgfrec",             KSTAT_DATA_UINT64 },
        { "pgin",               KSTAT_DATA_UINT64 },
        { "pgpgin",             KSTAT_DATA_UINT64 },
        { "pgout",              KSTAT_DATA_UINT64 },
        { "pgpgout",            KSTAT_DATA_UINT64 },
        { "swapin",             KSTAT_DATA_UINT64 },
        { "pgswapin",           KSTAT_DATA_UINT64 },
        { "swapout",            KSTAT_DATA_UINT64 },
        { "pgswapout",          KSTAT_DATA_UINT64 },
        { "zfod",               KSTAT_DATA_UINT64 },
        { "dfree",              KSTAT_DATA_UINT64 },
        { "scan",               KSTAT_DATA_UINT64 },
        { "rev",                KSTAT_DATA_UINT64 },
        { "hat_fault",          KSTAT_DATA_UINT64 },
        { "as_fault",           KSTAT_DATA_UINT64 },
        { "maj_fault",          KSTAT_DATA_UINT64 },
        { "cow_fault",          KSTAT_DATA_UINT64 },
        { "prot_fault",         KSTAT_DATA_UINT64 },
        { "softlock",           KSTAT_DATA_UINT64 },
        { "kernel_asflt",       KSTAT_DATA_UINT64 },
        { "pgrrun",             KSTAT_DATA_UINT64 },
        { "execpgin",           KSTAT_DATA_UINT64 },
        { "execpgout",          KSTAT_DATA_UINT64 },
        { "execfree",           KSTAT_DATA_UINT64 },
        { "anonpgin",           KSTAT_DATA_UINT64 },
```

```
        { "anonpgout",          KSTAT_DATA_UINT64 },
        { "anonfree",           KSTAT_DATA_UINT64 },
        { "fspgin",             KSTAT_DATA_UINT64 },
        { "fspgout",            KSTAT_DATA_UINT64 },
        { "fsfree",             KSTAT_DATA_UINT64 },
};
______unchanged_portion_omitted_

/*
 * Bind a thread to a CPU as requested.
 */
int
cpu_bind_thread(kthread_id_t tp, processorid_t bind, processorid_t *obind,
    int *error)
{
        processorid_t   binding;
        cpu_t           *cp = NULL;

        ASSERT(MUTEX_HELD(&cpu_lock));
        ASSERT(MUTEX_HELD(&ttoproc(tp)->p_lock));

        thread_lock(tp);

        /*
         * Record old binding, but change the obind, which was initialized
         * to PBIND_NONE, only if this thread has a binding.  This avoids
         * reporting PBIND_NONE for a process when some LWPs are bound.
         */
        binding = tp->t_bind_cpu;
        if (binding != PBIND_NONE)
                *obind = binding;       /* record old binding */

        switch (bind) {
        case PBIND_QUERY:
                /* Just return the old binding */
                thread_unlock(tp);
                return (0);

        case PBIND_QUERY_TYPE:
                /* Return the binding type */
                *obind = TB_CPU_IS_SOFT(tp) ? PBIND_SOFT : PBIND_HARD;
                thread_unlock(tp);
                return (0);

        case PBIND_SOFT:
                /*
                 *  Set soft binding for this thread and return the actual
                 *  binding
                 */
                TB_CPU_SOFT_SET(tp);
                thread_unlock(tp);
                return (0);

        case PBIND_HARD:
                /*
                 *  Set hard binding for this thread and return the actual
                 *  binding
                 */
                TB_CPU_HARD_SET(tp);
                thread_unlock(tp);
                return (0);

        default:
                break;
        }
        /*
         * If this thread/LWP cannot be bound because of permission
         * problems, just note that and return success so that the
         * other threads/LWPs will be bound.  This is the way
         * processor_bind() is defined to work.
         *
         * Binding will get EPERM if the thread is of system class
         * or hasprocperm() fails.
         */
        if (tp->t_cid == 0 || !hasprocperm(tp->t_cred, CRED())) {
                *error = EPERM;
                thread_unlock(tp);
                return (0);
        }

        binding = bind;
        if (binding != PBIND_NONE) {
                cp = cpu_get((processorid_t)binding);
                /*
                 * Make sure binding is valid and is in right partition.
                 */
                if (cp == NULL || tp->t_cpupart != cp->cpu_part) {
                        *error = EINVAL;
                        thread_unlock(tp);
                        return (0);
                }
        }
        tp->t_bind_cpu = binding;       /* set new binding */

        /*
         * If there is no system-set reason for affinity, set
         * the t_bound_cpu field to reflect the binding.
         */
        if (tp->t_affinitycnt == 0) {
                if (binding == PBIND_NONE) {
                        /*
                         * We may need to adjust disp_max_unbound_pri
                         * since we're becoming unbound.
                         */
                        disp_adjust_unbound_pri(tp);

                        tp->t_bound_cpu = NULL; /* set new binding */

                        /*
                         * Move thread to lgroup with strongest affinity
                         * after unbinding
                         */
                        if (tp->t_lgrp_affinity)
                                lgrp_move_thread(tp,
                                    lgrp_choose(tp, tp->t_cpupart), 1);

                        if (tp->t_state == TS_ONPROC &&
                            tp->t_cpu->cpu_part != tp->t_cpupart)
                                cpu_surrender(tp);
                } else {
                        lpl_t   *lpl;

                        tp->t_bound_cpu = cp;
                        ASSERT(cp->cpu_lpl != NULL);

                        /*
                         * Set home to lgroup with most affinity containing CPU
                         * that thread is being bound or minimum bounding
                         * lgroup if no affinities set
                         */
                        if (tp->t_lgrp_affinity)
```

```
                                lpl = lgrp_affinity_best(tp, tp->t_cpupart,
                                    LGRP_NONE, B_FALSE);
                        else
                                lpl = cp->cpu_lpl;

                        if (tp->t_lpl != lpl) {
                                /* can't grab cpu_lock */
                                lgrp_move_thread(tp, lpl, 1);
                        }

                        /*
                         * Make the thread switch to the bound CPU.
                         * If the thread is runnable, we need to
                         * requeue it even if t_cpu is already set
                         * to the right CPU, since it may be on a
                         * kpreempt queue and need to move to a local
                         * queue.  We could check t_disp_queue to
                         * avoid unnecessary overhead if it's already
                         * on the right queue, but since this isn't
                         * a performance-critical operation it doesn't
                         * seem worth the extra code and complexity.
                         *
                         * If the thread is weakbound to the cpu then it will
                         * resist the new binding request until the weak
                         * binding drops.  The cpu_surrender or requeueing
                         * below could be skipped in such cases (since it
                         * will have no effect), but that would require
                         * thread_allowmigrate to acquire thread_lock so
                         * we'll take the very occasional hit here instead.
                         */
                        if (tp->t_state == TS_ONPROC) {
                                cpu_surrender(tp);
                        } else if (tp->t_state == TS_RUN) {
                                cpu_t *ocp = tp->t_cpu;

                                (void) dispdeq(tp);
                                setbackdq(tp);
                                /*
                                 * On the bound CPU's disp queue now.
                                 * Either on the bound CPU's disp queue now,
                                 * or swapped out or on the swap queue.
                                 */
                                ASSERT(tp->t_disp_queue == cp->cpu_disp ||
                                    tp->t_weakbound_cpu == ocp);
                                    tp->t_weakbound_cpu == ocp ||
                                    (tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ))
                                    != TS_LOAD);
                        }
                }
        }

        /*
         * Our binding has changed; set TP_CHANGEBIND.
         */
        tp->t_proc_flag |= TP_CHANGEBIND;
        aston(tp);

        thread_unlock(tp);

        return (0);
}
```
______***unchanged_portion_omitted_***

```
static int
cpu_vm_stats_ks_update(kstat_t *ksp, int rw)
{
        cpu_t *cp = (cpu_t *)ksp->ks_private;
        struct cpu_vm_stats_ks_data *cvskd;
        cpu_vm_stats_t *cvs;

        if (rw == KSTAT_WRITE)
                return (EACCES);

        cvs = &cp->cpu_stats.vm;
        cvskd = ksp->ks_data;

        bcopy(&cpu_vm_stats_ks_data_template, ksp->ks_data,
            sizeof (cpu_vm_stats_ks_data_template));
        cvskd->pgrec.value.ui64 = cvs->pgrec;
        cvskd->pgfrec.value.ui64 = cvs->pgfrec;
        cvskd->pgin.value.ui64 = cvs->pgin;
        cvskd->pgpgin.value.ui64 = cvs->pgpgin;
        cvskd->pgout.value.ui64 = cvs->pgout;
        cvskd->pgpgout.value.ui64 = cvs->pgpgout;
        cvskd->swapin.value.ui64 = cvs->swapin;
        cvskd->pgswapin.value.ui64 = cvs->pgswapin;
        cvskd->swapout.value.ui64 = cvs->swapout;
        cvskd->pgswapout.value.ui64 = cvs->pgswapout;
        cvskd->zfod.value.ui64 = cvs->zfod;
        cvskd->dfree.value.ui64 = cvs->dfree;
        cvskd->scan.value.ui64 = cvs->scan;
        cvskd->rev.value.ui64 = cvs->rev;
        cvskd->hat_fault.value.ui64 = cvs->hat_fault;
        cvskd->as_fault.value.ui64 = cvs->as_fault;
        cvskd->maj_fault.value.ui64 = cvs->maj_fault;
        cvskd->cow_fault.value.ui64 = cvs->cow_fault;
        cvskd->prot_fault.value.ui64 = cvs->prot_fault;
        cvskd->softlock.value.ui64 = cvs->softlock;
        cvskd->kernel_asflt.value.ui64 = cvs->kernel_asflt;
        cvskd->pgrrun.value.ui64 = cvs->pgrrun;
        cvskd->execpgin.value.ui64 = cvs->execpgin;
        cvskd->execpgout.value.ui64 = cvs->execpgout;
        cvskd->execfree.value.ui64 = cvs->execfree;
        cvskd->anonpgin.value.ui64 = cvs->anonpgin;
        cvskd->anonpgout.value.ui64 = cvs->anonpgout;
        cvskd->anonfree.value.ui64 = cvs->anonfree;
        cvskd->fspgin.value.ui64 = cvs->fspgin;
        cvskd->fspgout.value.ui64 = cvs->fspgout;
        cvskd->fsfree.value.ui64 = cvs->fsfree;

        return (0);
}

static int
cpu_stat_ks_update(kstat_t *ksp, int rw)
{
        cpu_stat_t *cso;
        cpu_t *cp;
        int i;
        hrtime_t msnsecs[NCMSTATES];

        cso = (cpu_stat_t *)ksp->ks_data;
        cp = (cpu_t *)ksp->ks_private;

        if (rw == KSTAT_WRITE)
                return (EACCES);

        /*
         * Read CPU mstate, but compare with the last values we
         * received to make sure that the returned kstats never
         * decrease.
         */
```

```

        get_cpu_mstate(cp, msnsecs);
        msnsecs[CMS_IDLE] = NSEC_TO_TICK(msnsecs[CMS_IDLE]);
        msnsecs[CMS_USER] = NSEC_TO_TICK(msnsecs[CMS_USER]);
        msnsecs[CMS_SYSTEM] = NSEC_TO_TICK(msnsecs[CMS_SYSTEM]);
        if (cso->cpu_sysinfo.cpu[CPU_IDLE] < msnsecs[CMS_IDLE])
                cso->cpu_sysinfo.cpu[CPU_IDLE] = msnsecs[CMS_IDLE];
        if (cso->cpu_sysinfo.cpu[CPU_USER] < msnsecs[CMS_USER])
                cso->cpu_sysinfo.cpu[CPU_USER] = msnsecs[CMS_USER];
        if (cso->cpu_sysinfo.cpu[CPU_KERNEL] < msnsecs[CMS_SYSTEM])
                cso->cpu_sysinfo.cpu[CPU_KERNEL] = msnsecs[CMS_SYSTEM];
        cso->cpu_sysinfo.cpu[CPU_WAIT]  = 0;
        cso->cpu_sysinfo.wait[W_IO]     = 0;
        cso->cpu_sysinfo.wait[W_SWAP]   = 0;
        cso->cpu_sysinfo.wait[W_PIO]    = 0;
        cso->cpu_sysinfo.bread          = CPU_STATS(cp, sys.bread);
        cso->cpu_sysinfo.bwrite         = CPU_STATS(cp, sys.bwrite);
        cso->cpu_sysinfo.lread          = CPU_STATS(cp, sys.lread);
        cso->cpu_sysinfo.lwrite         = CPU_STATS(cp, sys.lwrite);
        cso->cpu_sysinfo.phread         = CPU_STATS(cp, sys.phread);
        cso->cpu_sysinfo.phwrite        = CPU_STATS(cp, sys.phwrite);
        cso->cpu_sysinfo.pswitch        = CPU_STATS(cp, sys.pswitch);
        cso->cpu_sysinfo.trap           = CPU_STATS(cp, sys.trap);
        cso->cpu_sysinfo.intr           = 0;
        for (i = 0; i < PIL_MAX; i++)
                cso->cpu_sysinfo.intr += CPU_STATS(cp, sys.intr[i]);
        cso->cpu_sysinfo.syscall        = CPU_STATS(cp, sys.syscall);
        cso->cpu_sysinfo.sysread        = CPU_STATS(cp, sys.sysread);
        cso->cpu_sysinfo.syswrite       = CPU_STATS(cp, sys.syswrite);
        cso->cpu_sysinfo.sysfork        = CPU_STATS(cp, sys.sysfork);
        cso->cpu_sysinfo.sysvfork       = CPU_STATS(cp, sys.sysvfork);
        cso->cpu_sysinfo.sysexec        = CPU_STATS(cp, sys.sysexec);
        cso->cpu_sysinfo.readch         = CPU_STATS(cp, sys.readch);
        cso->cpu_sysinfo.writech        = CPU_STATS(cp, sys.writech);
        cso->cpu_sysinfo.rcvint         = CPU_STATS(cp, sys.rcvint);
        cso->cpu_sysinfo.xmtint         = CPU_STATS(cp, sys.xmtint);
        cso->cpu_sysinfo.mdmint         = CPU_STATS(cp, sys.mdmint);
        cso->cpu_sysinfo.rawch          = CPU_STATS(cp, sys.rawch);
        cso->cpu_sysinfo.canch          = CPU_STATS(cp, sys.canch);
        cso->cpu_sysinfo.outch          = CPU_STATS(cp, sys.outch);
        cso->cpu_sysinfo.msg            = CPU_STATS(cp, sys.msg);
        cso->cpu_sysinfo.sema           = CPU_STATS(cp, sys.sema);
        cso->cpu_sysinfo.namei          = CPU_STATS(cp, sys.namei);
        cso->cpu_sysinfo.ufsiget        = CPU_STATS(cp, sys.ufsiget);
        cso->cpu_sysinfo.ufsdirblk      = CPU_STATS(cp, sys.ufsdirblk);
        cso->cpu_sysinfo.ufsipage       = CPU_STATS(cp, sys.ufsipage);
        cso->cpu_sysinfo.ufsinopage     = CPU_STATS(cp, sys.ufsinopage);
        cso->cpu_sysinfo.inodeovf       = 0;
        cso->cpu_sysinfo.fileovf        = 0;
        cso->cpu_sysinfo.procovf        = CPU_STATS(cp, sys.procovf);
        cso->cpu_sysinfo.intrthread     = 0;
        for (i = 0; i < LOCK_LEVEL - 1; i++)
                cso->cpu_sysinfo.intrthread += CPU_STATS(cp, sys.intr[i]);
        cso->cpu_sysinfo.intrblk        = CPU_STATS(cp, sys.intrblk);
        cso->cpu_sysinfo.idlethread     = CPU_STATS(cp, sys.idlethread);
        cso->cpu_sysinfo.inv_swtch      = CPU_STATS(cp, sys.inv_swtch);
        cso->cpu_sysinfo.nthreads       = CPU_STATS(cp, sys.nthreads);
        cso->cpu_sysinfo.cpumigrate     = CPU_STATS(cp, sys.cpumigrate);
        cso->cpu_sysinfo.xcalls         = CPU_STATS(cp, sys.xcalls);
        cso->cpu_sysinfo.mutex_adenters = CPU_STATS(cp, sys.mutex_adenters);
        cso->cpu_sysinfo.rw_rdfails     = CPU_STATS(cp, sys.rw_rdfails);
        cso->cpu_sysinfo.rw_wrfails     = CPU_STATS(cp, sys.rw_wrfails);
        cso->cpu_sysinfo.modload        = CPU_STATS(cp, sys.modload);
        cso->cpu_sysinfo.modunload      = CPU_STATS(cp, sys.modunload);
        cso->cpu_sysinfo.bawrite        = CPU_STATS(cp, sys.bawrite);
        cso->cpu_sysinfo.rw_enters      = 0;
        cso->cpu_sysinfo.win_uo_cnt     = 0;
        cso->cpu_sysinfo.win_uu_cnt     = 0;
        cso->cpu_sysinfo.win_so_cnt     = 0;
        cso->cpu_sysinfo.win_su_cnt     = 0;
        cso->cpu_sysinfo.win_suo_cnt    = 0;

        cso->cpu_syswait.iowait         = CPU_STATS(cp, sys.iowait);
        cso->cpu_syswait.swap           = 0;
        cso->cpu_syswait.physio         = 0;

        cso->cpu_vminfo.pgrec           = CPU_STATS(cp, vm.pgrec);
        cso->cpu_vminfo.pgfrec          = CPU_STATS(cp, vm.pgfrec);
        cso->cpu_vminfo.pgin            = CPU_STATS(cp, vm.pgin);
        cso->cpu_vminfo.pgpgin          = CPU_STATS(cp, vm.pgpgin);
        cso->cpu_vminfo.pgout           = CPU_STATS(cp, vm.pgout);
        cso->cpu_vminfo.pgpgout         = CPU_STATS(cp, vm.pgpgout);
        cso->cpu_vminfo.swapin          = CPU_STATS(cp, vm.swapin);
        cso->cpu_vminfo.pgswapin        = CPU_STATS(cp, vm.pgswapin);
        cso->cpu_vminfo.swapout         = CPU_STATS(cp, vm.swapout);
        cso->cpu_vminfo.pgswapout       = CPU_STATS(cp, vm.pgswapout);
        cso->cpu_vminfo.zfod            = CPU_STATS(cp, vm.zfod);
        cso->cpu_vminfo.dfree           = CPU_STATS(cp, vm.dfree);
        cso->cpu_vminfo.scan            = CPU_STATS(cp, vm.scan);
        cso->cpu_vminfo.rev             = CPU_STATS(cp, vm.rev);
        cso->cpu_vminfo.hat_fault       = CPU_STATS(cp, vm.hat_fault);
        cso->cpu_vminfo.as_fault        = CPU_STATS(cp, vm.as_fault);
        cso->cpu_vminfo.maj_fault       = CPU_STATS(cp, vm.maj_fault);
        cso->cpu_vminfo.cow_fault       = CPU_STATS(cp, vm.cow_fault);
        cso->cpu_vminfo.prot_fault      = CPU_STATS(cp, vm.prot_fault);
        cso->cpu_vminfo.softlock        = CPU_STATS(cp, vm.softlock);
        cso->cpu_vminfo.kernel_asflt    = CPU_STATS(cp, vm.kernel_asflt);
        cso->cpu_vminfo.pgrrun          = CPU_STATS(cp, vm.pgrrun);
        cso->cpu_vminfo.execpgin        = CPU_STATS(cp, vm.execpgin);
        cso->cpu_vminfo.execpgout       = CPU_STATS(cp, vm.execpgout);
        cso->cpu_vminfo.execfree        = CPU_STATS(cp, vm.execfree);
        cso->cpu_vminfo.anonpgin        = CPU_STATS(cp, vm.anonpgin);
        cso->cpu_vminfo.anonpgout       = CPU_STATS(cp, vm.anonpgout);
        cso->cpu_vminfo.anonfree        = CPU_STATS(cp, vm.anonfree);
        cso->cpu_vminfo.fspgin          = CPU_STATS(cp, vm.fspgin);
        cso->cpu_vminfo.fspgout         = CPU_STATS(cp, vm.fspgout);
        cso->cpu_vminfo.fsfree          = CPU_STATS(cp, vm.fsfree);

        return (0);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   15482 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/panic.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 */

/*
 * When the operating system detects that it is in an invalid state, a panic
 * is initiated in order to minimize potential damage to user data and to
 * facilitate debugging.  There are three major tasks to be performed in
 * a system panic: recording information about the panic in memory (and thus
 * making it part of the crash dump), synchronizing the file systems to
 * preserve user file data, and generating the crash dump.  We define the
 * system to be in one of four states with respect to the panic code:
 *
 * CALM    - the state of the system prior to any thread initiating a panic
 *
 * QUIESCE - the state of the system when the first thread to initiate
 *           a system panic records information about the cause of the panic
 *           and renders the system quiescent by stopping other processors
 *
 * SYNC    - the state of the system when we synchronize the file systems
 * DUMP    - the state when we generate the crash dump.
 *
 * The transitions between these states are irreversible: once we begin
 * panicking, we only make one attempt to perform the actions associated with
 * each state.
 *
 * The panic code itself must be re-entrant because actions taken during any
 * state may lead to another system panic.  Additionally, any Solaris
 * thread may initiate a panic at any time, and so we must have synchronization
 * between threads which attempt to initiate a state transition simultaneously.
 * The panic code makes use of a special locking primitive, a trigger, to
 * perform this synchronization.  A trigger is simply a word which is set
 * atomically and can only be set once.  We declare three triggers, one for
 * each transition between the four states.  When a thread enters the panic
 * code it attempts to set each trigger; if it fails it moves on to the
 * next trigger.  A special case is the first trigger: if two threads race
 * to perform the transition to QUIESCE, the losing thread may execute before
```

```
 * the winner has a chance to stop its CPU.  To solve this problem, we have
 * the loser look ahead to see if any other triggers are set; if not, it
 * presumes a panic is underway and simply spins.  Unfortunately, since we
 * are panicking, it is not possible to know this with absolute certainty.
 *
 * There are two common reasons for re-entering the panic code once a panic
 * has been initiated: (1) after we debug_enter() at the end of QUIESCE,
 * the operator may type "sync" instead of "go", and the PROM's sync callback
 * routine will invoke panic(); (2) if the clock routine decides that sync
 * or dump is not making progress, it will invoke panic() to force a timeout.
 * The design assumes that a third possibility, another thread causing an
 * unrelated panic while sync or dump is still underway, is extremely unlikely.
 * If this situation occurs, we may end up triggering dump while sync is
 * still in progress.  This third case is considered extremely unlikely because
 * all other CPUs are stopped and low-level interrupts have been blocked.
 *
 * The panic code is entered via a call directly to the vpanic() function,
 * or its varargs wrappers panic() and cmn_err(9F).  The vpanic routine
 * is implemented in assembly language to record the current machine
 * registers, attempt to set the trigger for the QUIESCE state, and
 * if successful, switch stacks on to the panic_stack before calling into
 * the common panicsys() routine.  The first thread to initiate a panic
 * is allowed to make use of the reserved panic_stack so that executing
 * the panic code itself does not overwrite valuable data on that thread's
 * stack *ahead* of the current stack pointer.  This data will be preserved
 * in the crash dump and may prove invaluable in determining what this
 * thread has previously been doing.  The first thread, saved in panic_thread,
 * is also responsible for stopping the other CPUs as quickly as possible,
 * and then setting the various panic_* variables.  Most important among
 * these is panicstr, which allows threads to subsequently bypass held
 * locks so that we can proceed without ever blocking.  We must stop the
 * other CPUs *prior* to setting panicstr in case threads running there are
 * currently spinning to acquire a lock; we want that state to be preserved.
 * Every thread which initiates a panic has its T_PANIC flag set so we can
 * identify all such threads in the crash dump.
 *
 * The panic_thread is also allowed to make use of the special memory buffer
 * panicbuf, which on machines with appropriate hardware is preserved across
 * reboots.  We allow the panic_thread to store its register set and panic
 * message in this buffer, so even if we fail to obtain a crash dump we will
 * be able to examine the machine after reboot and determine some of the
 * state at the time of the panic.  If we do get a dump, the panic buffer
 * data is structured so that a debugger can easily consume the information
 * therein (see <sys/panic.h>).
 *
 * Each platform or architecture is required to implement the functions
 * panic_savetrap() to record trap-specific information to panicbuf,
 * panic_saveregs() to record a register set to panicbuf, panic_stopcpus()
 * to halt all CPUs but the panicking CPU, panic_quiesce_hw() to perform
 * miscellaneous platform-specific tasks *after* panicstr is set,
 * panic_showtrap() to print trap-specific information to the console,
 * and panic_dump_hw() to perform platform tasks prior to calling dumpsys().
 *
 * A Note on Word Formation, courtesy of the Oxford Guide to English Usage:
 *
 * Words ending in -c interpose k before suffixes which otherwise would
 * indicate a soft c, and thus the verb and adjective forms of 'panic' are
 * spelled "panicked", "panicking", and "panicky" respectively.  Use of
 * the ill-conceived "panicing" and "panic'd" is discouraged.
 */

#include <sys/types.h>
#include <sys/varargs.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/cpuvar.h>
```

```c
#include <sys/thread.h>
#include <sys/t_lock.h>
#include <sys/cred.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/uadmin.h>
#include <sys/callb.h>
#include <sys/vfs.h>
#include <sys/log.h>
#include <sys/disp.h>
#include <sys/param.h>
#include <sys/dumphdr.h>
#include <sys/ftrace.h>
#include <sys/reboot.h>
#include <sys/debug.h>
#include <sys/stack.h>
#include <sys/spl.h>
#include <sys/errorq.h>
#include <sys/panic.h>
#include <sys/fm/util.h>
#include <sys/clock_impl.h>

/*
 * Panic variables which are set once during the QUIESCE state by the
 * first thread to initiate a panic.  These are examined by post-mortem
 * debugging tools; the inconsistent use of 'panic' versus 'panic_' in
 * the variable naming is historical and allows legacy tools to work.
 */
#pragma align STACK_ALIGN(panic_stack)
char panic_stack[PANICSTKSIZE];         /* reserved stack for panic_thread */
kthread_t *panic_thread;                /* first thread to call panicsys() */
cpu_t panic_cpu;                        /* cpu from first call to panicsys() */
label_t panic_regs;                     /* setjmp label from panic_thread */
label_t panic_pcb;                      /* t_pcb at time of panic */
struct regs *panic_reg;                 /* regs struct from first panicsys() */
char *volatile panicstr;                /* format string to first panicsys() */
va_list panicargs;                      /* arguments to first panicsys() */
clock_t panic_lbolt;                    /* lbolt at time of panic */
int64_t panic_lbolt64;                  /* lbolt64 at time of panic */
hrtime_t panic_hrtime;                  /* hrtime at time of panic */
timespec_t panic_hrestime;              /* hrestime at time of panic */
int panic_ipl;                          /* ipl on panic_cpu at time of panic */
ushort_t panic_schedflag;               /* t_schedflag for panic_thread */
cpu_t *panic_bound_cpu;                 /* t_bound_cpu for panic_thread */
char panic_preempt;                     /* t_preempt for panic_thread */

/*
 * Panic variables which can be set via /etc/system or patched while
 * the system is in operation.  Again, the stupid names are historic.
 */
char *panic_bootstr = NULL;             /* mdboot string to use after panic */
int panic_bootfcn = AD_BOOT;            /* mdboot function to use after panic */
int halt_on_panic = 0;                  /* halt after dump instead of reboot? */
int nopanicdebug = 0;                   /* reboot instead of call debugger? */
int in_sync = 0;                        /* skip vfs_syncall() and just dump? */

/*
 * The do_polled_io flag is set by the panic code to inform the SCSI subsystem
 * to use polled mode instead of interrupt-driven i/o.
 */
int do_polled_io = 0;

/*
 * The panic_forced flag is set by the uadmin A_DUMP code to inform the
 * panic subsystem that it should not attempt an initial debug_enter.
 */
```

```c
int panic_forced = 0;

/*
 * Triggers for panic state transitions:
 */
int panic_quiesce;                      /* trigger for CALM    -> QUIESCE */
int panic_sync;                         /* trigger for QUIESCE -> SYNC */
int panic_dump;                         /* trigger for SYNC    -> DUMP */

/*
 * Variable signifying quiesce(9E) is in progress.
 */
volatile int quiesce_active = 0;

void
panicsys(const char *format, va_list alist, struct regs *rp, int on_panic_stack)
{
        int s = spl8();
        kthread_t *t = curthread;
        cpu_t *cp = CPU;

        caddr_t intr_stack = NULL;
        uint_t intr_actv;

        ushort_t schedflag = t->t_schedflag;
        cpu_t *bound_cpu = t->t_bound_cpu;
        char preempt = t->t_preempt;
        label_t pcb = t->t_pcb;

        (void) setjmp(&t->t_pcb);
        t->t_flag |= T_PANIC;

        t->t_schedflag |= TS_DONT_SWAP;
        t->t_bound_cpu = cp;
        t->t_preempt++;

        panic_enter_hw(s);

        /*
         * If we're on the interrupt stack and an interrupt thread is available
         * in this CPU's pool, preserve the interrupt stack by detaching an
         * interrupt thread and making its stack the intr_stack.
         */
        if (CPU_ON_INTR(cp) && cp->cpu_intr_thread != NULL) {
                kthread_t *it = cp->cpu_intr_thread;

                intr_stack = cp->cpu_intr_stack;
                intr_actv = cp->cpu_intr_actv;

                cp->cpu_intr_stack = thread_stk_init(it->t_stk);
                cp->cpu_intr_thread = it->t_link;

                /*
                 * Clear only the high level bits of cpu_intr_actv.
                 * We want to indicate that high-level interrupts are
                 * not active without destroying the low-level interrupt
                 * information stored there.
                 */
                cp->cpu_intr_actv &= ((1 << (LOCK_LEVEL + 1)) - 1);
        }

        /*
         * Record one-time panic information and quiesce the other CPUs.
         * Then print out the panic message and stack trace.
         */
        if (on_panic_stack) {
                panic_data_t *pdp = (panic_data_t *)panicbuf;

                pdp->pd_version = PANICBUFVERS;
                pdp->pd_msgoff = sizeof (panic_data_t) - sizeof (panic_nv_t);

                (void) strncpy(pdp->pd_uuid, dump_get_uuid(),
                    sizeof (pdp->pd_uuid));

                if (t->t_panic_trap != NULL)
                        panic_savetrap(pdp, t->t_panic_trap);
                else
                        panic_saveregs(pdp, rp);

                (void) vsnprintf(&panicbuf[pdp->pd_msgoff],
                    PANICBUFSIZE - pdp->pd_msgoff, format, alist);

                /*
                 * Call into the platform code to stop the other CPUs.
                 * We currently have all interrupts blocked, and expect that
                 * the platform code will lower ipl only as far as needed to
                 * perform cross-calls, and will acquire as *few* locks as is
                 * possible -- panicstr is not set so we can still deadlock.
                 */
                panic_stopcpus(cp, t, s);

                panicstr = (char *)format;
                va_copy(panicargs, alist);
                panic_lbolt = LBOLT_NO_ACCOUNT;
                panic_lbolt64 = LBOLT_NO_ACCOUNT64;
                panic_hrestime = hrestime;
                panic_hrtime = gethrtime_waitfree();
                panic_thread = t;
                panic_regs = t->t_pcb;
                panic_reg = rp;
                panic_cpu = *cp;
                panic_ipl = spltoipl(s);
                panic_schedflag = schedflag;
                panic_bound_cpu = bound_cpu;
                panic_preempt = preempt;
                panic_pcb = pcb;

                if (intr_stack != NULL) {
                        panic_cpu.cpu_intr_stack = intr_stack;
                        panic_cpu.cpu_intr_actv = intr_actv;
                }

                /*
                 * Lower ipl to 10 to keep clock() from running, but allow
                 * keyboard interrupts to enter the debugger.  These callbacks
                 * are executed with panicstr set so they can bypass locks.
                 */
                splx(ipltospl(CLOCK_LEVEL));
                panic_quiesce_hw(pdp);
                (void) FTRACE_STOP();
                (void) callb_execute_class(CB_CL_PANIC, NULL);

                if (log_intrq != NULL)
                        log_flushq(log_intrq);

                /*
                 * If log_consq has been initialized and syslogd has started,
                 * print any messages in log_consq that haven't been consumed.
                 */
                if (log_consq != NULL && log_consq != log_backlogq)
                        log_printq(log_consq);
```

```
                fm_banner();

#if defined(__x86)
                /*
                 * A hypervisor panic originates outside of Solaris, so we
                 * don't want to prepend the panic message with misleading
                 * pointers from within Solaris.
                 */
                if (!IN_XPV_PANIC())
#endif
                        printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id,
                            (void *)t);
                vprintf(format, alist);
                printf("\n\n");

                if (t->t_panic_trap != NULL) {
                        panic_showtrap(t->t_panic_trap);
                        printf("\n");
                }

                traceregs(rp);
                printf("\n");

                if (((boothowto & RB_DEBUG) || obpdebug) &&
                    !nopanicdebug && !panic_forced) {
                        if (dumpvp != NULL) {
                                debug_enter("panic: entering debugger "
                                    "(continue to save dump)");
                        } else {
                                debug_enter("panic: entering debugger "
                                    "(no dump device, continue to reboot)");
                        }
                }

        } else if (panic_dump != 0 || panic_sync != 0 || panicstr != NULL) {
                printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id, (void *)t);
                vprintf(format, alist);
                printf("\n");
        } else
                goto spin;

        /*
         * Prior to performing sync or dump, we make sure that do_polled_io is
         * set, but we'll leave ipl at 10; deadman(), a CY_HIGH_LEVEL cyclic,
         * will re-enter panic if we are not making progress with sync or dump.
         */

        /*
         * Sync the filesystems.  Reset t_cred if not set because much of
         * the filesystem code depends on CRED() being valid.
         */
        if (!in_sync && panic_trigger(&panic_sync)) {
                if (t->t_cred == NULL)
                        t->t_cred = kcred;
                splx(ipltospl(CLOCK_LEVEL));
                do_polled_io = 1;
                vfs_syncall();
        }

        /*
         * Take the crash dump.  If the dump trigger is already set, try to
         * enter the debugger again before rebooting the system.
         */
        if (panic_trigger(&panic_dump)) {
                panic_dump_hw(s);
                splx(ipltospl(CLOCK_LEVEL));
```

```
		errorq_panic();
		do_polled_io = 1;
		dumpsys();
	} else if (((boothowto & RB_DEBUG) || obpdebug) && !nopanicdebug) {
		debug_enter("panic: entering debugger (continue to reboot)");
	} else
		printf("dump aborted: please record the above information!\n");

	if (halt_on_panic)
		mdboot(A_REBOOT, AD_HALT, NULL, B_FALSE);
	else
		mdboot(A_REBOOT, panic_bootfcn, panic_bootstr, B_FALSE);
spin:
	/*
	 * Restore ipl to at most CLOCK_LEVEL so we don’t end up spinning
	 * and unable to jump into the debugger.
	 */
	splx(MIN(s, ipltospl(CLOCK_LEVEL)));
	for (;;)
		;
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
    2754 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/sched.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/cpuvar.h>
#include <sys/var.h>
#include <sys/tuneable.h>
#include <sys/cmn_err.h>
#include <sys/buf.h>
#include <sys/disp.h>
#include <sys/vmsystm.h>
#include <sys/vmparam.h>
#include <sys/class.h>
#include <sys/vtrace.h>
#include <sys/modctl.h>
#include <sys/debug.h>
#include <sys/tnf_probe.h>
#include <sys/procfs.h>

#include <vm/seg.h>
#include <vm/seg_kp.h>
#include <vm/as.h>
#include <vm/rm.h>
#include <vm/seg_kmem.h>
#include <sys/callb.h>

/*
 * The swapper sleeps on runout when there is no one to swap in.
 * It sleeps on runin when it could not find space to swap someone
 * in or after swapping someone in.
 */
```

```
char    runout;
char    runin;
char    wake_sched;     /* flag tells clock to wake swapper on next tick */
char    wake_sched_sec; /* flag tells clock to wake swapper after a second */

/*
 * The swapper swaps processes to reduce memory demand and runs
 * when avefree < desfree.  The swapper resorts to SOFTSWAP when
 * avefree < desfree which results in swapping out all processes
 * sleeping for more than maxslp seconds.  HARDSWAP occurs when the
 * system is on the verge of thrashing and this results in swapping
 * out runnable threads or threads sleeping for less than maxslp secs.
 *
 * The swapper runs through all the active processes in the system
 * and invokes the scheduling class specific swapin/swapout routine
 * for every thread in the process to obtain an effective priority
 * for the process.  A priority of -1 implies that the thread isn't
 * swappable.  This effective priority is used to find the most
 * eligible process to swapout or swapin.
 *
 * NOTE:  Threads which have been swapped are not linked on any
 *        queue and their dispatcher lock points at the "swapped_lock".
 *
 * Processes containing threads with the TS_DONT_SWAP flag set cannot be
 * swapped out immediately by the swapper.  This is due to the fact that
 * such threads may be holding locks which may be needed by the swapper
 * to push its pages out.  The TS_SWAPENQ flag is set on such threads
 * to prevent them running in user mode.  When such threads reach a
 * safe point (i.e., are not holding any locks - CL_TRAPRET), they
 * queue themseleves onto the swap queue which is processed by the
 * swapper.  This results in reducing memory demand when the system
 * is desparate for memory as the thread can't run in user mode.
 *
 * The swap queue consists of threads, linked via t_link, which are
 * haven't been swapped, are runnable but not on the run queue.  The
 * swap queue is protected by the "swapped_lock".  The dispatcher
 * lock (t_lockp) of all threads on the swap queue points at the
 * "swapped_lock".  Thus, the entire queue and/or threads on the
 * queue can be locked by acquiring "swapped_lock".
 */
static kthread_t *tswap_queue;
extern disp_lock_t swapped_lock; /* protects swap queue and threads on it */

int     maxslp = 0;
pgcnt_t avefree;        /* 5 sec moving average of free memory */
pgcnt_t avefree30;      /* 30 sec moving average of free memory */

/*
 * Minimum size used to decide if sufficient memory is available
 * before a process is swapped in.  This is necessary since in most
 * cases the actual size of a process (p_swrss) being swapped in
 * is usually 2 pages (kernel stack pages).  This is due to the fact
 * almost all user pages of a process are stolen by pageout before
 * the swapper decides to swapout it out.
 */
int     min_procsize = 12;

static int      swapin(proc_t *);
static int      swapout(proc_t *, uint_t *, int);
static void     process_swap_queue();

#ifdef __sparc
extern void lwp_swapin(kthread_t *);
#endif /* __sparc */

/*
```

```
 * Counters to keep track of the number of swapins or swapouts.
 */
uint_t tot_swapped_in, tot_swapped_out;
uint_t softswap, hardswap, swapqswap;

/*
 * Macro to determine if a process is eligble to be swapped.
 */
#define not_swappable(p)                                        \
        (((p)->p_flag & SSYS) || (p)->p_stat == SIDL ||         \
            (p)->p_stat == SZOMB || (p)->p_as == NULL ||        \
            (p)->p_as == &kas)

/*
 * Memory scheduler.
 */
void
sched()
{
        kthread_id_t    t;
        pri_t           proc_pri;
        pri_t           thread_pri;
        pri_t           swapin_pri;
        int             desperate;
        pgcnt_t         needs;
        int             divisor;
        proc_t          *prp;
        proc_t          *swapout_prp;
        proc_t          *swapin_prp;
        spgcnt_t        avail;
        int             chosen_pri;
        time_t          swapout_time;
        time_t          swapin_proc_time;
        callb_cpr_t     cprinfo;
        kmutex_t        swap_cpr_lock;

        mutex_init(&swap_cpr_lock, NULL, MUTEX_DEFAULT, NULL);
        CALLB_CPR_INIT(&cprinfo, &swap_cpr_lock, callb_generic_cpr, "sched");
        if (maxslp == 0)
                maxslp = MAXSLP;
loop:
        needs = 0;
        desperate = 0;

        for (;;) {
        swapin_pri = v.v_nglobpris;
        swapin_prp = NULL;
        chosen_pri = -1;

        process_swap_queue();

        /*
         * Set desperate if
         *      1.  At least 2 runnable processes (on average).
         *      2.  Short (5 sec) and longer (30 sec) average is less
         *          than minfree and desfree respectively.
         *      3.  Pagein + pageout rate is excessive.
         */
                if (avenrun[0] >= 2 * FSCALE &&
                    (MAX(avefree, avefree30) < desfree) &&
                    (pginrate + pgoutrate > maxpgio || avefree < minfree)) {
                TRACE_4(TR_FAC_SCHED, TR_DESPERATE,
                    "desp:avefree: %d, avefree30: %d, freemem: %d"
                    " pginrate: %d\n", avefree, avefree30, freemem, pginrate);
                desperate = 1;
                goto unload;
```

```
        }

        /*
         * Search list of processes to swapin and swapout deadwood.
         */
        swapin_proc_time = 0;
top:
        mutex_enter(&pidlock);
        for (prp = practive; prp != NULL; prp = prp->p_next) {
                if (not_swappable(prp))
                        continue;

                /*
                 * Look at processes with at least one swapped lwp.
                 */
                if (prp->p_swapcnt) {
                        time_t proc_time;

                        /*
                         * Higher priority processes are good candidates
                         * to swapin.
                         */
                        mutex_enter(&prp->p_lock);
                        proc_pri = -1;
                        t = prp->p_tlist;
                        proc_time = 0;
                        do {
                                if (t->t_schedflag & TS_LOAD)
                                        continue;

                                thread_lock(t);
                                thread_pri = CL_SWAPIN(t, 0);
                                thread_unlock(t);

                                if (t->t_stime - proc_time > 0)
                                        proc_time = t->t_stime;
                                if (thread_pri > proc_pri)
                                        proc_pri = thread_pri;
                        } while ((t = t->t_forw) != prp->p_tlist);
                        mutex_exit(&prp->p_lock);

                        if (proc_pri == -1)
                                continue;

                        TRACE_3(TR_FAC_SCHED, TR_CHOOSE_SWAPIN,
                            "prp %p epri %d proc_time %d",
                            prp, proc_pri, proc_time);

                        /*
                         * Swapin processes with a high effective priority.
                         */
                        if (swapin_prp == NULL || proc_pri > chosen_pri) {
                                swapin_prp = prp;
                                chosen_pri = proc_pri;
                                swapin_pri = proc_pri;
                                swapin_proc_time = proc_time;
                        }
                } else {
                        /*
                         * No need to soft swap if we have sufficient
                         * memory.
                         */
                        if (avefree > desfree ||
                            avefree < desfree && freemem > desfree)
                                continue;
                        /*
                         * Skip processes that are exiting
                         * or whose address spaces are locked.
                         */
                        mutex_enter(&prp->p_lock);
                        if ((prp->p_flag & SEXITING) ||
                            (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
                                mutex_exit(&prp->p_lock);
                                continue;
                        }

                        /*
                         * Softswapping to kick out deadwood.
                         */
                        proc_pri = -1;
                        t = prp->p_tlist;
                        do {
                                if ((t->t_schedflag & (TS_SWAPENQ |
                                    TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
                                        continue;

                                thread_lock(t);
                                thread_pri = CL_SWAPOUT(t, SOFTSWAP);
                                thread_unlock(t);
                                if (thread_pri > proc_pri)
                                        proc_pri = thread_pri;
                        } while ((t = t->t_forw) != prp->p_tlist);

                        if (proc_pri != -1) {
                                uint_t swrss;

                                mutex_exit(&pidlock);

                                TRACE_1(TR_FAC_SCHED, TR_SOFTSWAP,
                                    "softswap:prp %p", prp);

                                (void) swapout(prp, &swrss, SOFTSWAP);
                                softswap++;
                                prp->p_swrss += swrss;
                                mutex_exit(&prp->p_lock);
                                goto top;
                        }
                        mutex_exit(&prp->p_lock);
                }
        }
        if (swapin_prp != NULL)
                mutex_enter(&swapin_prp->p_lock);
        mutex_exit(&pidlock);

        if (swapin_prp == NULL) {
                TRACE_3(TR_FAC_SCHED, TR_RUNOUT,
                "schedrunout:runout nswapped: %d, avefree: %ld freemem: %ld",
                    nswapped, avefree, freemem);

                t = curthread;
                thread_lock(t);
                runout++;
                t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
                t->t_whystop = PR_SUSPENDED;
                t->t_whatstop = SUSPEND_NORMAL;
                (void) new_mstate(t, LMS_SLEEP);
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_BEGIN(&cprinfo);
                mutex_exit(&swap_cpr_lock);
                thread_stop(t);         /* change state and drop lock */
                swtch();
```

```
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
                mutex_exit(&swap_cpr_lock);
                goto loop;
        }

        /*
         * Decide how deserving this process is to be brought in.
         * Needs is an estimate of how much core the process will
         * need.  If the process has been out for a while, then we
         * will bring it in with 1/2 the core needed, otherwise
         * we are conservative.
         */
        divisor = 1;
        swapout_time = (ddi_get_lbolt() - swapin_proc_time) / hz;
        if (swapout_time > maxslp / 2)
                divisor = 2;

        needs = MIN(swapin_prp->p_swrss, lotsfree);
        needs = MAX(needs, min_procsize);
        needs = needs / divisor;

        /*
         * Use freemem, since we want processes to be swapped
         * in quickly.
         */
        avail = freemem - deficit;
        if (avail > (spgcnt_t)needs) {
                deficit += needs;

                TRACE_2(TR_FAC_SCHED, TR_SWAPIN_VALUES,
                    "swapin_values: prp %p needs %lu", swapin_prp, needs);

                if (swapin(swapin_prp)) {
                        mutex_exit(&swapin_prp->p_lock);
                        goto loop;
                }
                deficit -= MIN(needs, deficit);
                mutex_exit(&swapin_prp->p_lock);
        } else {
                mutex_exit(&swapin_prp->p_lock);
                /*
                 * If deficit is high, too many processes have been
                 * swapped in so wait a sec before attempting to
                 * swapin more.
                 */
                if (freemem > needs) {
                        TRACE_2(TR_FAC_SCHED, TR_HIGH_DEFICIT,
                            "deficit: prp %p needs %lu", swapin_prp, needs);
                        goto block;
                }
        }

        TRACE_2(TR_FAC_SCHED, TR_UNLOAD,
            "unload: prp %p needs %lu", swapin_prp, needs);

unload:
                        /*
                         * Unload all unloadable modules, free all other memory
                         * resources we can find, then look for a thread to
                         * hardswap.
         * resources we can find, then look for a thread to hardswap.
                         */
                        modreap();
                        segkp_cache_free();
        swapout_prp = NULL;
        mutex_enter(&pidlock);
        for (prp = practive; prp != NULL; prp = prp->p_next) {

                /*
                 * No need to soft swap if we have sufficient
                 * memory.
                 */
                if (not_swappable(prp))
                        continue;

                if (avefree > minfree ||
                    avefree < minfree && freemem > desfree) {
                        swapout_prp = NULL;
                        break;
                }

                /*
                 * Skip processes that are exiting
                 * or whose address spaces are locked.
                 */
                mutex_enter(&prp->p_lock);
                if ((prp->p_flag & SEXITING) ||
                    (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
                        mutex_exit(&prp->p_lock);
                        continue;
                }

                proc_pri = -1;
                t = prp->p_tlist;
                do {
                        if ((t->t_schedflag & (TS_SWAPENQ |
                            TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
                                continue;

                        thread_lock(t);
                        thread_pri = CL_SWAPOUT(t, HARDSWAP);
                        thread_unlock(t);
                        if (thread_pri > proc_pri)
                                proc_pri = thread_pri;
                } while ((t = t->t_forw) != prp->p_tlist);

                mutex_exit(&prp->p_lock);
                if (proc_pri == -1)
                        continue;

                /*
                 * Swapout processes sleeping with a lower priority
                 * than the one currently being swapped in, if any.
                 */
                if (swapin_prp == NULL || swapin_pri > proc_pri) {
                        TRACE_2(TR_FAC_SCHED, TR_CHOOSE_SWAPOUT,
                            "hardswap: prp %p needs %lu", prp, needs);

                        if (swapout_prp == NULL || proc_pri < chosen_pri) {
                                swapout_prp = prp;
                                chosen_pri = proc_pri;
                        }
                }
        }

        /*
         * Acquire the "p_lock" before dropping "pidlock"
         * to prevent the proc structure from being freed
         * if the process exits before swapout completes.
         */
```

```
        if (swapout_prp != NULL)
                mutex_enter(&swapout_prp->p_lock);
        mutex_exit(&pidlock);

        if ((prp = swapout_prp) != NULL) {
                uint_t swrss = 0;
                int swapped;

                swapped = swapout(prp, &swrss, HARDSWAP);
                if (swapped) {
                        /*
                         * If desperate, we want to give the space obtained
                         * by swapping this process out to processes in core,
                         * so we give them a chance by increasing deficit.
                         */
                        prp->p_swrss += swrss;
                        if (desperate)
                                deficit += MIN(prp->p_swrss, lotsfree);
                        hardswap++;
                }
                mutex_exit(&swapout_prp->p_lock);

                if (swapped)
                        goto loop;
        }

        /*
         * Delay for 1 second and look again later.
         */
        TRACE_3(TR_FAC_SCHED, TR_RUNIN,
            "schedrunin:runin nswapped: %d, avefree: %ld freemem: %ld",
            nswapped, avefree, freemem);

block:
                t = curthread;
                thread_lock(t);
        runin++;
                t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
                t->t_whystop = PR_SUSPENDED;
                t->t_whatstop = SUSPEND_NORMAL;
                (void) new_mstate(t, LMS_SLEEP);
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_BEGIN(&cprinfo);
                mutex_exit(&swap_cpr_lock);
                thread_stop(t);         /* change to stop state and drop lock */
                swtch();
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
                mutex_exit(&swap_cpr_lock);
        goto loop;
}

/*
 * Remove the specified thread from the swap queue.
 */
static void
swapdeq(kthread_id_t tp)
{
        kthread_id_t *tpp;

        ASSERT(THREAD_LOCK_HELD(tp));
        ASSERT(tp->t_schedflag & TS_ON_SWAPQ);

        tpp = &tswap_queue;
        for (;;) {
                ASSERT(*tpp != NULL);
```

```
		if (*tpp == tp)
			break;
		tpp = &(*tpp)->t_link;
	}
	*tpp = tp->t_link;
	tp->t_schedflag &= ~TS_ON_SWAPQ;
}

/*
 * Swap in lwps.  Returns nonzero on success (i.e., if at least one lwp is
 * swapped in) and 0 on failure.
 */
static int
swapin(proc_t *pp)
{
	kthread_id_t tp;
	int err;
	int num_swapped_in = 0;
	struct cpu *cpup = CPU;
	pri_t thread_pri;

	ASSERT(MUTEX_HELD(&pp->p_lock));
	ASSERT(pp->p_swapcnt);

top:
	tp = pp->p_tlist;
	do {
		/*
		 * Only swapin eligible lwps (specified by the scheduling
		 * class) which are unloaded and ready to run.
		 */
		thread_lock(tp);
		thread_pri = CL_SWAPIN(tp, 0);
		if (thread_pri != -1 && tp->t_state == TS_RUN &&
		    (tp->t_schedflag & TS_LOAD) == 0) {
			size_t stack_size;
			pgcnt_t stack_pages;

			ASSERT((tp->t_schedflag & TS_ON_SWAPQ) == 0);

			thread_unlock(tp);
			/*
			 * Now drop the p_lock since the stack needs
			 * to brought in.
			 */
			mutex_exit(&pp->p_lock);

			stack_size = swapsize(tp->t_swap);
			stack_pages = btopr(stack_size);
			/* Kernel probe */
			TNF_PROBE_4(swapin_lwp, "vm swap swapin", /* CSTYLED */,
			    tnf_pid,		pid,		pp->p_pid,
			    tnf_lwpid,		lwpid,		tp->t_tid,
			    tnf_kthread_id,	tid,		tp,
			    tnf_ulong,		page_count,	stack_pages);

			rw_enter(&kas.a_lock, RW_READER);
			err = segkp_fault(segkp->s_as->a_hat, segkp,
			    tp->t_swap, stack_size, F_SOFTLOCK, S_OTHER);
			rw_exit(&kas.a_lock);

			/*
			 * Re-acquire the p_lock.
			 */
			mutex_enter(&pp->p_lock);
			if (err) {
```

```
				num_swapped_in = 0;
				break;
			} else {
#ifdef __sparc
				lwp_swapin(tp);
#endif /* __sparc */
				CPU_STATS_ADDQ(cpup, vm, swapin, 1);
				CPU_STATS_ADDQ(cpup, vm, pgswapin,
				    stack_pages);

				pp->p_swapcnt--;
				pp->p_swrss -= stack_pages;

				thread_lock(tp);
				tp->t_schedflag |= TS_LOAD;
				dq_sruninc(tp);

				/* set swapin time */
				tp->t_stime = ddi_get_lbolt();
				thread_unlock(tp);

				nswapped--;
				tot_swapped_in++;
				num_swapped_in++;

				TRACE_2(TR_FAC_SCHED, TR_SWAPIN,
				    "swapin: pp %p stack_pages %lu",
				    pp, stack_pages);
				goto top;
			}
		}
		thread_unlock(tp);
	} while ((tp = tp->t_forw) != pp->p_tlist);
	return (num_swapped_in);
}

/*
 * Swap out lwps.  Returns nonzero on success (i.e., if at least one lwp is
 * swapped out) and 0 on failure.
 */
static int
swapout(proc_t *pp, uint_t *swrss, int swapflags)
{
	kthread_id_t tp;
	pgcnt_t ws_pages = 0;
	int err;
	int swapped_lwps = 0;
	struct as *as = pp->p_as;
	struct cpu *cpup = CPU;
	pri_t thread_pri;

	ASSERT(MUTEX_HELD(&pp->p_lock));

	if (pp->p_flag & SEXITING)
		return (0);

top:
	tp = pp->p_tlist;
	do {
		klwp_t *lwp = ttolwp(tp);

		/*
		 * Swapout eligible lwps (specified by the scheduling
		 * class) which don't have TS_DONT_SWAP set.  Set the
		 * "intent to swap" flag (TS_SWAPENQ) on threads
		 * which have TS_DONT_SWAP set so that they can be
		 * swapped if and when they reach a safe point.
		 */
		thread_lock(tp);
		thread_pri = CL_SWAPOUT(tp, swapflags);
		if (thread_pri != -1) {
			if (tp->t_schedflag & TS_DONT_SWAP) {
				tp->t_schedflag |= TS_SWAPENQ;
				tp->t_trapret = 1;
				aston(tp);
			} else {
				pgcnt_t stack_pages;
				size_t stack_size;

				ASSERT((tp->t_schedflag &
				    (TS_DONT_SWAP | TS_LOAD)) == TS_LOAD);

				if (lock_try(&tp->t_lock)) {
					/*
					 * Remove thread from the swap_queue.
					 */
					if (tp->t_schedflag & TS_ON_SWAPQ) {
						ASSERT(!(tp->t_schedflag &
						    TS_SWAPENQ));
						swapdeq(tp);
					} else if (tp->t_state == TS_RUN)
						dq_srundec(tp);

					tp->t_schedflag &=
					    ~(TS_LOAD | TS_SWAPENQ);
					lock_clear(&tp->t_lock);

					/*
					 * Set swapout time if the thread isn't
					 * sleeping.
					 */
					if (tp->t_state != TS_SLEEP)
						tp->t_stime = ddi_get_lbolt();
					thread_unlock(tp);

					nswapped++;
					tot_swapped_out++;

					lwp->lwp_ru.nswap++;

					/*
					 * Now drop the p_lock since the
					 * stack needs to pushed out.
					 */
					mutex_exit(&pp->p_lock);

					stack_size = swapsize(tp->t_swap);
					stack_pages = btopr(stack_size);
					ws_pages += stack_pages;
					/* Kernel probe */
					TNF_PROBE_4(swapout_lwp,
					    "vm swap swapout",
					    /* CSTYLED */,
					    tnf_pid, pid, pp->p_pid,
					    tnf_lwpid, lwpid, tp->t_tid,
					    tnf_kthread_id, tid, tp,
					    tnf_ulong, page_count,
					    stack_pages);

					rw_enter(&kas.a_lock, RW_READER);
					err = segkp_fault(segkp->s_as->a_hat,
					    segkp, tp->t_swap, stack_size,
```

```
					    F_SOFTUNLOCK, S_WRITE);
					rw_exit(&kas.a_lock);

					if (err) {
						cmn_err(CE_PANIC,
						    "swapout: segkp_fault "
						    "failed err: %d", err);
					}
					CPU_STATS_ADDQ(cpup,
					    vm, pgswapout, stack_pages);

					mutex_enter(&pp->p_lock);
					pp->p_swapcnt++;
					swapped_lwps++;
					goto top;
				}
			}
		}
		thread_unlock(tp);
	} while ((tp = tp->t_forw) != pp->p_tlist);

	/*
	 * Unload address space when all lwps are swapped out.
	 */
	if (pp->p_swapcnt == pp->p_lwpcnt) {
		size_t as_size = 0;

		/*
		 * Avoid invoking as_swapout() if the process has
		 * no MMU resources since pageout will eventually
		 * steal pages belonging to this address space.  This
		 * saves CPU cycles as the number of pages that are
		 * potentially freed or pushed out by the segment
		 * swapout operation is very small.
		 */
		if (rm_asrss(pp->p_as) != 0)
			as_size = as_swapout(as);

		CPU_STATS_ADDQ(cpup, vm, pgswapout, btop(as_size));
		CPU_STATS_ADDQ(cpup, vm, swapout, 1);
		ws_pages += btop(as_size);

		TRACE_2(TR_FAC_SCHED, TR_SWAPOUT,
		    "swapout: pp %p pages_pushed %lu", pp, ws_pages);
		/* Kernel probe */
		TNF_PROBE_2(swapout_process, "vm swap swapout", /* CSTYLED */,
		    tnf_pid,    pid,            pp->p_pid,
		    tnf_ulong,  page_count,     ws_pages);
	}
	*swrss = ws_pages;
	return (swapped_lwps);
}

void
swapout_lwp(klwp_t *lwp)
{
	kthread_id_t tp = curthread;

	ASSERT(curthread == lwptot(lwp));

	/*
	 * Don't insert the thread onto the swap queue if
	 * sufficient memory is available.
	 */
	if (avefree > desfree || avefree < desfree && freemem > desfree) {
		thread_lock(tp);
		tp->t_schedflag &= ~TS_SWAPENQ;
		thread_unlock(tp);
		return;
	}

	/*
	 * Lock the thread, then move it to the swapped queue from the
	 * onproc queue and set its state to be TS_RUN.
	 */
	thread_lock(tp);
	ASSERT(tp->t_state == TS_ONPROC);
	if (tp->t_schedflag & TS_SWAPENQ) {
		tp->t_schedflag &= ~TS_SWAPENQ;

		/*
		 * Set the state of this thread to be runnable
		 * and move it from the onproc queue to the swap queue.
		 */
		disp_swapped_enq(tp);

		/*
		 * Insert the thread onto the swap queue.
		 */
		tp->t_link = tswap_queue;
		tswap_queue = tp;
		tp->t_schedflag |= TS_ON_SWAPQ;

		thread_unlock_nopreempt(tp);

		TRACE_1(TR_FAC_SCHED, TR_SWAPOUT_LWP, "swapout_lwp:%x", lwp);

		swtch();
	} else {
		thread_unlock(tp);
	}
}

/*
 * Swap all threads on the swap queue.
 */
static void
process_swap_queue(void)
{
	kthread_id_t tp;
	uint_t ws_pages;
	proc_t *pp;
	struct cpu *cpup = CPU;
	klwp_t *lwp;
	int err;

	if (tswap_queue == NULL)
		return;

	/*
	 * Acquire the "swapped_lock" which locks the swap queue,
	 * and unload the stacks of all threads on it.
	 */
	disp_lock_enter(&swapped_lock);
	while ((tp = tswap_queue) != NULL) {
		pgcnt_t stack_pages;
		size_t stack_size;

		tswap_queue = tp->t_link;
		tp->t_link = NULL;

		/*
```

```
		 * Drop the "dispatcher lock" before acquiring "t_lock"
		 * to avoid spinning on it since the thread at the front
		 * of the swap queue could be pinned before giving up
		 * its "t_lock" in resume.
		 */
		disp_lock_exit(&swapped_lock);
		lock_set(&tp->t_lock);

		/*
		 * Now, re-acquire the "swapped_lock".  Acquiring this lock
		 * results in locking the thread since its dispatcher lock
		 * (t_lockp) is the "swapped_lock".
		 */
		disp_lock_enter(&swapped_lock);
		ASSERT(tp->t_state == TS_RUN);
		ASSERT(tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ));

		tp->t_schedflag &= ~(TS_LOAD | TS_ON_SWAPQ);
		tp->t_stime = ddi_get_lbolt();		/* swapout time */
		disp_lock_exit(&swapped_lock);
		lock_clear(&tp->t_lock);

		lwp = ttolwp(tp);
		lwp->lwp_ru.nswap++;

		pp = ttoproc(tp);
		stack_size = swapsize(tp->t_swap);
		stack_pages = btopr(stack_size);

		/* Kernel probe */
		TNF_PROBE_4(swapout_lwp, "vm swap swapout", /* CSTYLED */,
		    tnf_pid,		pid,		pp->p_pid,
		    tnf_lwpid,		lwpid,		tp->t_tid,
		    tnf_kthread_id,	tid,		tp,
		    tnf_ulong,		page_count,	stack_pages);

		rw_enter(&kas.a_lock, RW_READER);
		err = segkp_fault(segkp->s_as->a_hat, segkp, tp->t_swap,
		    stack_size, F_SOFTUNLOCK, S_WRITE);
		rw_exit(&kas.a_lock);

		if (err) {
			cmn_err(CE_PANIC,
			"process_swap_list: segkp_fault failed err: %d", err);
		}
		CPU_STATS_ADDQ(cpup, vm, pgswapout, stack_pages);

		nswapped++;
		tot_swapped_out++;
		swapqswap++;

		/*
		 * Don't need p_lock since the swapper is the only
		 * thread which increments/decrements p_swapcnt and p_swrss.
		 */
		ws_pages = stack_pages;
		pp->p_swapcnt++;

		TRACE_1(TR_FAC_SCHED, TR_SWAPQ_LWP, "swaplist: pp %p", pp);

		/*
		 * Unload address space when all lwps are swapped out.
		 */
		if (pp->p_swapcnt == pp->p_lwpcnt) {
			size_t as_size = 0;
			if (rm_asrss(pp->p_as) != 0)
				as_size = as_swapout(pp->p_as);

			CPU_STATS_ADDQ(cpup, vm, pgswapout,
			    btop(as_size));
			CPU_STATS_ADDQ(cpup, vm, swapout, 1);

			ws_pages += btop(as_size);

			TRACE_2(TR_FAC_SCHED, TR_SWAPQ_PROC,
			    "swaplist_proc: pp %p pages_pushed: %lu",
			    pp, ws_pages);
			/* Kernel probe */
			TNF_PROBE_2(swapout_process, "vm swap swapout",
			    /* CSTYLED */,
			    tnf_pid,	pid,		pp->p_pid,
			    tnf_ulong,	page_count,	ws_pages);
		}
		pp->p_swrss += ws_pages;
		disp_lock_enter(&swapped_lock);
	}
	disp_lock_exit(&swapped_lock);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   39428 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/timers.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*
 * Real time profiling interval timer expired:
 * Increment microstate counters for each lwp in the process
 * and ensure that running lwps are kicked into the kernel.
 * If time is not set up to reload, then just return.
 * Else compute next time timer should go off which is > current time,
 * as above.
 */
static void
realprofexpire(void *arg)
{
        struct proc *p = arg;
        kthread_t *t;

        mutex_enter(&p->p_lock);
        if (p->p_rprof_cyclic == CYCLIC_NONE ||
            (t = p->p_tlist) == NULL) {
                mutex_exit(&p->p_lock);
                return;
        }
        do {
                int mstate;

                /*
                 * Attempt to allocate the SIGPROF buffer, but don't sleep.
                 */
                if (t->t_rprof == NULL)
                        t->t_rprof = kmem_zalloc(sizeof (struct rprof),
                            KM_NOSLEEP);
                if (t->t_rprof == NULL)
                        continue;

                thread_lock(t);
                switch (t->t_state) {
                case TS_SLEEP:
                        /*
                         * Don't touch the lwp is it is swapped out.
                         */
                        if (!(t->t_schedflag & TS_LOAD)) {
                                mstate = LMS_SLEEP;
                                break;
                        }
                        switch (mstate = ttolwp(t)->lwp_mstate.ms_prev) {
                        case LMS_TFAULT:
                        case LMS_DFAULT:
                        case LMS_KFAULT:
                        case LMS_USER_LOCK:
                                break;
                        default:
                                mstate = LMS_SLEEP;
                                break;
                        }
                        break;
                case TS_RUN:
                case TS_WAIT:
                        mstate = LMS_WAIT_CPU;
                        break;
                case TS_ONPROC:
                        switch (mstate = t->t_mstate) {
```

```
                        case LMS_USER:
                        case LMS_SYSTEM:
                        case LMS_TRAP:
                                break;
                        default:
                                mstate = LMS_SYSTEM;
                                break;
                        }
                        break;
                default:
                        mstate = t->t_mstate;
                        break;
                }
                t->t_rprof->rp_anystate = 1;
                t->t_rprof->rp_state[mstate]++;
                aston(t);
                /*
                 * force the thread into the kernel
                 * if it is not already there.
                 */
                if (t->t_state == TS_ONPROC && t->t_cpu != CPU)
                        poke_cpu(t->t_cpu->cpu_id);
                thread_unlock(t);
        } while ((t = t->t_forw) != p->p_tlist);

        mutex_exit(&p->p_lock);
}
______unchanged_portion_omitted_
```

***
****
   10025 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/waitq.c
6583 remove whole-process swapping
****
______unchanged_portion_omitted_***

```
/*
 * Put specified thread to specified wait queue without dropping thread's lock.
 * Returns 1 if thread was successfully placed on project's wait queue, or
 * 0 if wait queue is blocked.
 */
int
waitq_enqueue(waitq_t *wq, kthread_t *t)
{
        ASSERT(THREAD_LOCK_HELD(t));
        ASSERT(t->t_sleepq == NULL);
        ASSERT(t->t_waitq == NULL);
        ASSERT(t->t_link == NULL);

        disp_lock_enter_high(&wq->wq_lock);

        /*
         * Can't enqueue anything on a blocked wait queue
         */
        if (wq->wq_blocked) {
                disp_lock_exit_high(&wq->wq_lock);
                return (0);
        }

        /*
         * Mark the time when thread is placed on wait queue. The microstate
         * accounting code uses this timestamp to determine wait times.
         */
        t->t_waitrq = gethrtime_unscaled();

        /*
         * Mark thread as not swappable.  If necessary, it will get
         * swapped out when it returns to the userland.
         */
        t->t_schedflag |= TS_DONT_SWAP;
        DTRACE_SCHED1(cpucaps__sleep, kthread_t *, t);
        waitq_link(wq, t);

        THREAD_WAIT(t, &wq->wq_lock);
        return (1);
}
```

______unchanged_portion_omitted_

****
    7493 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/class.h
6583 remove whole-process swapping
****
______*unchanged_portion_omitted_*

```
typedef struct thread_ops {
        int     (*cl_enterclass)(kthread_t *, id_t, void *, cred_t *, void *);
        void    (*cl_exitclass)(void *);
        int     (*cl_canexit)(kthread_t *, cred_t *);
        int     (*cl_fork)(kthread_t *, kthread_t *, void *);
        void    (*cl_forkret)(kthread_t *, kthread_t *);
        void    (*cl_parmsget)(kthread_t *, void *);
        int     (*cl_parmsset)(kthread_t *, void *, id_t, cred_t *);
        void    (*cl_stop)(kthread_t *, int, int);
        void    (*cl_exit)(kthread_t *);
        void    (*cl_active)(kthread_t *);
        void    (*cl_inactive)(kthread_t *);
        pri_t   (*cl_swapin)(kthread_t *, int);
        pri_t   (*cl_swapout)(kthread_t *, int);
        void    (*cl_trapret)(kthread_t *);
        void    (*cl_preempt)(kthread_t *);
        void    (*cl_setrun)(kthread_t *);
        void    (*cl_sleep)(kthread_t *);
        void    (*cl_tick)(kthread_t *);
        void    (*cl_wakeup)(kthread_t *);
        int     (*cl_donice)(kthread_t *, cred_t *, int, int *);
        pri_t   (*cl_globpri)(kthread_t *);
        void    (*cl_set_process_group)(pid_t, pid_t, pid_t);
        void    (*cl_yield)(kthread_t *);
        int     (*cl_doprio)(kthread_t *, cred_t *, int, int *);
} thread_ops_t;
```

______*unchanged_portion_omitted_*

```
#define STATIC_SCHED            (krwlock_t *)0xffffffff
#define LOADABLE_SCHED(s)       ((s)->cl_lock != STATIC_SCHED)
#define SCHED_INSTALLED(s)      ((s)->cl_funcs != NULL)
#define ALLOCATED_SCHED(s)      ((s)->cl_lock != NULL)

#if defined(_KERNEL) || defined(_FAKE_KERNEL)

#define CLASS_KERNEL(cid)       ((cid) == syscid || (cid) == sysdccid)

extern int      nclass;         /* number of configured scheduling classes */
extern char     *defaultclass;  /* default class for newproc'd processes */
extern struct sclass sclass[];  /* the class table */
extern kmutex_t class_lock;     /* lock protecting class table */
extern int      loaded_classes; /* number of classes loaded */

extern pri_t    minclsyspri;
extern id_t     syscid;         /* system scheduling class ID */
extern id_t     sysdccid;       /* system duty-cycle scheduling class ID */
extern id_t     defaultcid;     /* "default" class id; see dispadmin(1M) */

extern int      alloc_cid(char *, id_t *);
extern int      scheduler_load(char *, sclass_t *);
extern int      getcid(char *, id_t *);
extern int      getcidbyname(char *, id_t *);
extern int      parmsin(pcparms_t *, pc_vaparms_t *);
extern int      parmsout(pcparms_t *, pc_vaparms_t *);
extern int      parmsset(pcparms_t *, kthread_t *);
extern void     parmsget(kthread_t *, pcparms_t *);
extern int      vaparmsout(char *, pcparms_t *, pc_vaparms_t *, uio_seg_t);

#endif
```

```
#define CL_ADMIN(clp, uaddr, reqpcredp) \
        (*(clp)->cl_funcs->sclass.cl_admin)(uaddr, reqpcredp)

#define CL_ENTERCLASS(t, cid, clparmsp, credp, bufp) \
        (sclass[cid].cl_funcs->thread.cl_enterclass) (t, cid, \
            (void *)clparmsp, credp, bufp)

#define CL_EXITCLASS(cid, clprocp)\
        (sclass[cid].cl_funcs->thread.cl_exitclass) ((void *)clprocp)

#define CL_CANEXIT(t, cr)       (*(t)->t_clfuncs->cl_canexit)(t, cr)

#define CL_FORK(tp, ct, bufp)   (*(tp)->t_clfuncs->cl_fork)(tp, ct, bufp)

#define CL_FORKRET(t, ct)       (*(t)->t_clfuncs->cl_forkret)(t, ct)

#define CL_GETCLINFO(clp, clinfop) \
        (*(clp)->cl_funcs->sclass.cl_getclinfo)((void *)clinfop)

#define CL_GETCLPRI(clp, clprip) \
        (*(clp)->cl_funcs->sclass.cl_getclpri)(clprip)

#define CL_PARMSGET(t, clparmsp) \
        (*(t)->t_clfuncs->cl_parmsget)(t, (void *)clparmsp)

#define CL_PARMSIN(clp, clparmsp) \
        (clp)->cl_funcs->sclass.cl_parmsin((void *)clparmsp)

#define CL_PARMSOUT(clp, clparmsp, vaparmsp) \
        (clp)->cl_funcs->sclass.cl_parmsout((void *)clparmsp, vaparmsp)

#define CL_VAPARMSIN(clp, clparmsp, vaparmsp) \
        (clp)->cl_funcs->sclass.cl_vaparmsin((void *)clparmsp, vaparmsp)

#define CL_VAPARMSOUT(clp, clparmsp, vaparmsp) \
        (clp)->cl_funcs->sclass.cl_vaparmsout((void *)clparmsp, vaparmsp)

#define CL_PARMSSET(t, clparmsp, cid, curpcredp) \
        (*(t)->t_clfuncs->cl_parmsset)(t, (void *)clparmsp, cid, curpcredp)

#define CL_PREEMPT(tp)          (*(tp)->t_clfuncs->cl_preempt)(tp)

#define CL_SETRUN(tp)           (*(tp)->t_clfuncs->cl_setrun)(tp)

#define CL_SLEEP(tp)            (*(tp)->t_clfuncs->cl_sleep)(tp)

#define CL_STOP(t, why, what)   (*(t)->t_clfuncs->cl_stop)(t, why, what)

#define CL_EXIT(t)              (*(t)->t_clfuncs->cl_exit)(t)

#define CL_ACTIVE(t)            (*(t)->t_clfuncs->cl_active)(t)

#define CL_INACTIVE(t)          (*(t)->t_clfuncs->cl_inactive)(t)

#define CL_SWAPIN(t, flags)     (*(t)->t_clfuncs->cl_swapin)(t, flags)

#define CL_SWAPOUT(t, flags)    (*(t)->t_clfuncs->cl_swapout)(t, flags)

#define CL_TICK(t)              (*(t)->t_clfuncs->cl_tick)(t)

#define CL_TRAPRET(t)           (*(t)->t_clfuncs->cl_trapret)(t)

#define CL_WAKEUP(t)            (*(t)->t_clfuncs->cl_wakeup)(t)

#define CL_DONICE(t, cr, inc, ret) \
        (*(t)->t_clfuncs->cl_donice)(t, cr, inc, ret)

#define CL_DOPRIO(t, cr, inc, ret) \
        (*(t)->t_clfuncs->cl_doprio)(t, cr, inc, ret)

#define CL_GLOBPRI(t)           (*(t)->t_clfuncs->cl_globpri)(t)

#define CL_SET_PROCESS_GROUP(t, s, b, f) \
        (*(t)->t_clfuncs->cl_set_process_group)(s, b, f)

#define CL_YIELD(tp)            (*(tp)->t_clfuncs->cl_yield)(tp)

#define CL_ALLOC(pp, cid, flag) \
        (sclass[cid].cl_funcs->sclass.cl_alloc) (pp, flag)

#define CL_FREE(cid, bufp)      (sclass[cid].cl_funcs->sclass.cl_free) (bufp)

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
    5872 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/disp.h
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#if defined(_KERNEL) || defined(_FAKE_KERNEL)

#define MAXCLSYSPRI     99
#define MINCLSYSPRI     60


/*
 * Global scheduling variables.
 *      - See sys/cpuvar.h for CPU-local variables.
 */
extern int      nswapped;       /* number of swapped threads */
                                /* nswapped protected by swap_lock */

extern  pri_t   minclsyspri;    /* minimum level of any system class */
extern  pri_t   maxclsyspri;    /* maximum level of any system class */
extern  pri_t   intr_pri;       /* interrupt thread priority base level */

#endif  /* _KERNEL || _FAKE_KERNEL */
#if defined(_KERNEL)

/*
 * Minimum amount of time that a thread can remain runnable before it can
 * be stolen by another CPU (in nanoseconds).
 */
extern hrtime_t nosteal_nsec;

/*
 * Kernel preemption occurs if a higher-priority thread is runnable with
 * a priority at or above kpreemptpri.
 *
 * So that other processors can watch for such threads, a separate
 * dispatch queue with unbound work above kpreemptpri is maintained.
 * This is part of the CPU partition structure (cpupart_t).
 */
extern  pri_t   kpreemptpri;    /* level above which preemption takes place */

extern void             disp_kp_alloc(disp_t *, pri_t); /* allocate kp queue */
extern void             disp_kp_free(disp_t *);         /* free kp queue */

/*
 * Macro for use by scheduling classes to decide whether the thread is about
 * to be scheduled or not.  This returns the maximum run priority.
 */
#define DISP_MAXRUNPRI(t)       ((t)->t_disp_queue->disp_maxrunpri)

/*
 * Platform callbacks for various dispatcher operations
 *
 * idle_cpu() is invoked when a cpu goes idle, and has nothing to do.
 * disp_enq_thread() is invoked when a thread is placed on a run queue.
 */
extern void     (*idle_cpu)();
extern void     (*disp_enq_thread)(struct cpu *, int);


extern int              dispdeq(kthread_t *);
extern void             dispinit(void);
extern void             disp_add(sclass_t *);
extern int              intr_active(struct cpu *, int);
```

```
extern int              servicing_interrupt(void);
extern void             preempt(void);
extern void             setbackdq(kthread_t *);
extern void             setfrontdq(kthread_t *);
extern void             swtch(void);
extern void             swtch_to(kthread_t *);
extern void             swtch_from_zombie(void)
                                __NORETURN;
extern void             dq_sruninc(kthread_t *);
extern void             dq_srundec(kthread_t *);
extern void             cpu_rechoose(kthread_t *);
extern void             cpu_surrender(kthread_t *);
extern void             kpreempt(int);
extern struct cpu       *disp_lowpri_cpu(struct cpu *, struct lgrp_ld *, pri_t,
                            struct cpu *);
extern int              disp_bound_threads(struct cpu *, int);
extern int              disp_bound_anythreads(struct cpu *, int);
extern int              disp_bound_partition(struct cpu *, int);
extern void             disp_cpu_init(struct cpu *);
extern void             disp_cpu_fini(struct cpu *);
extern void             disp_cpu_inactive(struct cpu *);
extern void             disp_adjust_unbound_pri(kthread_t *);
extern void             resume(kthread_t *);
extern void             resume_from_intr(kthread_t *);
extern void             resume_from_zombie(kthread_t *)
                                __NORETURN;
extern void             disp_swapped_enq(kthread_t *);
extern int              disp_anywork(void);

#define KPREEMPT_SYNC           (-1)
#define kpreempt_disable()                              \
        {                                               \
                curthread->t_preempt++;                 \
                ASSERT(curthread->t_preempt >= 1);      \
        }
#define kpreempt_enable()                               \
        {                                               \
                ASSERT(curthread->t_preempt >= 1);      \
                if (--curthread->t_preempt == 0 &&      \
                    CPU->cpu_kprunrun)                  \
                        kpreempt(KPREEMPT_SYNC);        \
        }

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
   29368 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/proc.h
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

struct pool;
struct task;
struct zone;
struct brand;
struct corectl_path;
struct corectl_content;

/*
 * One structure allocated per active process.  Per-process data (user.h) is
 * also inside the proc structure.
 * One structure allocated per active process.  It contains all
 * data needed about the process while the process may be swapped
 * out.  Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 */
typedef struct  proc {
        /*
         * Fields requiring no explicit locking
         */
        struct  vnode *p_exec;          /* pointer to a.out vnode */
        struct  as *p_as;               /* process address space pointer */
        struct  plock *p_lockp;         /* ptr to proc struct's mutex lock */
        kmutex_t p_crlock;              /* lock for p_cred */
        struct  cred    *p_cred;        /* process credentials */
        /*
         * Fields protected by pidlock
         */
        int     p_swapcnt;              /* number of swapped out lwps */
        char    p_stat;                 /* status of process */
        char    p_wcode;                /* current wait code */
        ushort_t p_pidflag;             /* flags protected only by pidlock */
        int     p_wdata;                /* current wait return value */
        pid_t   p_ppid;                 /* process id of parent */
        struct  proc    *p_link;        /* forward link */
        struct  proc    *p_parent;      /* ptr to parent process */
        struct  proc    *p_child;       /* ptr to first child process */
        struct  proc    *p_sibling;     /* ptr to next sibling proc on chain */
        struct  proc    *p_psibling;    /* ptr to prev sibling proc on chain */
        struct  proc    *p_sibling_ns;  /* prt to siblings with new state */
        struct  proc    *p_child_ns;    /* prt to children with new state */
        struct  proc    *p_next;        /* active chain link next */
        struct  proc    *p_prev;        /* active chain link prev */
        struct  proc    *p_nextofkin;   /* gets accounting info at exit */
        struct  proc    *p_orphan;
        struct  proc    *p_nextorph;
        struct  proc    *p_pglink;      /* process group hash chain link next */
        struct  proc    *p_ppglink;     /* process group hash chain link prev */
        struct  sess    *p_sessp;       /* session information */
        struct  pid     *p_pidp;        /* process ID info */
        struct  pid     *p_pgidp;       /* process group ID info */
        /*
         * Fields protected by p_lock
         */
        kcondvar_t p_cv;                /* proc struct's condition variable */
        kcondvar_t p_flag_cv;
        kcondvar_t p_lwpexit;           /* waiting for some lwp to exit */
        kcondvar_t p_holdlwps;          /* process is waiting for its lwps */
                                        /* to to be held.  */
        uint_t  p_proc_flag;            /* /proc-related flags */
```

```
        uint_t  p_flag;                 /* protected while set. */
                                        /* flags defined below */
        clock_t p_utime;                /* user time, this process */
        clock_t p_stime;                /* system time, this process */
        clock_t p_cutime;               /* sum of children's user time */
        clock_t p_cstime;               /* sum of children's system time */
        avl_tree_t *p_segacct;          /* System V shared segment list */
        avl_tree_t *p_semacct;          /* System V semaphore undo list */
        caddr_t p_bssbase;              /* base addr of last bss below heap */
        caddr_t p_brkbase;              /* base addr of heap */
        size_t  p_brksize;              /* heap size in bytes */
        uint_t  p_brkpageszc;           /* preferred heap max page size code */
        /*
         * Per process signal stuff.
         */
        k_sigset_t p_sig;               /* signals pending to this process */
        k_sigset_t p_extsig;            /* signals sent from another contract */
        k_sigset_t p_ignore;            /* ignore when generated */
        k_sigset_t p_siginfo;           /* gets signal info with signal */
        void *p_sigfd;                  /* signalfd support state */
        struct sigqueue *p_sigqueue;    /* queued siginfo structures */
        struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
        struct sigqhdr *p_signhdr;      /* hdr to signotify structure pool */
        uchar_t p_stopsig;              /* jobcontrol stop signal */

        /*
         * Special per-process flag when set will fix misaligned memory
         * references.
         */
        char    p_fixalignment;

        /*
         * Per process lwp and kernel thread stuff
         */
        id_t    p_lwpid;                /* most recently allocated lwpid */
        int     p_lwpcnt;               /* number of lwps in this process */
        int     p_lwprcnt;              /* number of not stopped lwps */
        int     p_lwpdaemon;            /* number of TP_DAEMON lwps */
        int     p_lwpwait;              /* number of lwps in lwp_wait() */
        int     p_lwpdwait;             /* number of daemons in lwp_wait() */
        int     p_zombcnt;              /* number of zombie lwps */
        kthread_t *p_tlist;             /* circular list of threads */
        lwpdir_t *p_lwpdir;             /* thread (lwp) directory */
        lwpdir_t *p_lwpfree;            /* p_lwpdir free list */
        tidhash_t *p_tidhash;           /* tid (lwpid) lookup hash table */
        uint_t  p_lwpdir_sz;            /* number of p_lwpdir[] entries */
        uint_t  p_tidhash_sz;           /* number of p_tidhash[] entries */
        ret_tidhash_t *p_ret_tidhash;   /* retired tidhash hash tables */
        uint64_t p_lgrpset;             /* unprotected hint of set of lgrps */
                                        /* on which process has threads */
        volatile lgrp_id_t  p_t1_lgrpid; /* main's thread lgroup id */
        volatile lgrp_id_t  p_tr_lgrpid; /* text replica's lgroup id */
#if defined(_LP64)
        uintptr_t  p_lgrpres2;          /* reserved for lgrp migration */
#endif
        /*
         * /proc (process filesystem) debugger interface stuff.
         */
        k_sigset_t p_sigmask;           /* mask of traced signals (/proc) */
        k_fltset_t p_fltmask;           /* mask of traced faults (/proc) */
        struct vnode *p_trace;          /* pointer to primary /proc vnode */
        struct vnode *p_plist;          /* list of /proc vnodes for process */
        kthread_t *p_agenttp;           /* thread ptr for /proc agent lwp */
        avl_tree_t p_warea;             /* list of watched areas */
        avl_tree_t p_wpage;             /* remembered watched pages (vfork) */
        watched_page_t *p_wprot;        /* pages that need to have prot set */
        int     p_mapcnt;               /* number of active pr_mappage()s */
        kmutex_t p_maplock;             /* lock for pr_mappage() */
        struct  proc  *p_rlink;         /* linked list for server */
        kcondvar_t p_srwchan_cv;
        size_t  p_stksize;              /* process stack size in bytes */
        uint_t  p_stkpageszc;           /* preferred stack max page size code */

        /*
         * Microstate accounting, resource usage, and real-time profiling
         */
        hrtime_t p_mstart;              /* hi-res process start time */
        hrtime_t p_mterm;               /* hi-res process termination time */
        hrtime_t p_mlreal;              /* elapsed time sum over defunct lwps */
        hrtime_t p_acct[NMSTATES];      /* microstate sum over defunct lwps */
        hrtime_t p_cacct[NMSTATES];     /* microstate sum over child procs */
        struct lrusage p_ru;            /* lrusage sum over defunct lwps */
        struct lrusage p_cru;           /* lrusage sum over child procs */
        struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
        uintptr_t p_rprof_cyclic;       /* ITIMER_REALPROF cyclic */
        uint_t  p_defunct;              /* number of defunct lwps */
        /*
         * profiling. A lock is used in the event of multiple lwp’s
         * using the same profiling base/size.
         */
        kmutex_t p_pflock;              /* protects user profile arguments */
        struct prof p_prof;             /* profile arguments */

        /*
         * Doors.
         */
        door_pool_t             p_server_threads; /* common thread pool */
        struct door_node        *p_door_list;   /* active doors */
        struct door_node        *p_unref_list;
        kcondvar_t              p_unref_cv;
        char                    p_unref_thread; /* unref thread created */

        /*
         * Kernel probes
         */
        uchar_t                 p_tnf_flags;

        /*
         * Solaris Audit
         */
        struct p_audit_data     *p_audit_data; /* per process audit structure */

        pctxop_t        *p_pctx;

#if defined(__x86)
        /*
         * LDT support.
         */
        kmutex_t        p_ldtlock;      /* protects the following fields */
        user_desc_t     *p_ldt;         /* Pointer to private LDT */
        system_desc_t   p_ldt_desc;     /* segment descriptor for private LDT */
        ushort_t        p_ldtlimit;     /* highest selector used */
#endif
        size_t p_swrss;                 /* resident set size before last swap */
        struct aio      *p_aio;         /* pointer to async I/O struct */
        struct itimer   **p_itimer;     /* interval timers */
        timeout_id_t    p_alarmid;      /* alarm’s timeout id */
        caddr_t         p_usrstack;     /* top of the process stack */
        uint_t          p_stkprot;      /* stack memory protection */
        uint_t          p_datprot;      /* data memory protection */
        model_t         p_model;        /* data model determined at exec time */
        struct lwpchan_data *p_lcp;     /* lwpchan cache */
```

```
        kmutex_t        p_lcp_lock;     /* protects assignments to p_lcp */
        utrap_handler_t *p_utraps;      /* pointer to user trap handlers */
        struct corectl_path     *p_corefile;    /* pattern for core file */
        struct task     *p_task;        /* our containing task */
        struct proc     *p_taskprev;    /* ptr to previous process in task */
        struct proc     *p_tasknext;    /* ptr to next process in task */
        kmutex_t        p_sc_lock;      /* protects p_pagep */
        struct sc_page_ctl *p_pagep;    /* list of process’s shared pages */
        struct rctl_set *p_rctls;       /* resource controls for this process */
        rlim64_t        p_stk_ctl;      /* currently enforced stack size */
        rlim64_t        p_fsz_ctl;      /* currently enforced file size */
        rlim64_t        p_vmem_ctl;     /* currently enforced addr-space size */
        rlim64_t        p_fno_ctl;      /* currently enforced file-desc limit */
        pid_t           p_ancpid;       /* ancestor pid, used by exacct */
        struct itimerval p_realitimer;  /* real interval timer */
        timeout_id_t    p_itimerid;     /* real interval timer’s timeout id */
        struct corectl_content *p_content;      /* content of core file */

        avl_tree_t      p_ct_held;      /* held contracts */
        struct ct_equeue **p_ct_equeue; /* process-type event queues */

        struct cont_process *p_ct_process; /* process contract */
        list_node_t     p_ct_member;    /* process contract membership */
        sigqueue_t      *p_killsqp;     /* sigqueue pointer for SIGKILL */

        int             p_dtrace_probes; /* are there probes for this proc? */
        uint64_t        p_dtrace_count; /* number of DTrace tracepoints */
                                        /* (protected by P_PR_LOCK) */
        void            *p_dtrace_helpers; /* DTrace helpers, if any */
        struct pool     *p_pool;        /* pointer to containing pool */
        kcondvar_t      p_poolcv;       /* synchronization with pools */
        uint_t          p_poolcnt;      /* # threads inside pool barrier */
        uint_t          p_poolflag;     /* pool-related flags (see below) */
        uintptr_t       p_portcnt;      /* event ports counter */
        struct zone     *p_zone;        /* zone in which process lives */
        struct vnode    *p_execdir;     /* directory that p_exec came from */
        struct brand    *p_brand;       /* process’s brand  */
        void            *p_brand_data;  /* per-process brand state */

        /* additional lock to protect p_sessp (but not its contents) */
        kmutex_t p_splock;
        rctl_qty_t      p_locked_mem;   /* locked memory charged to proc */
                                        /* protected by p_lock */
        rctl_qty_t      p_crypto_mem;   /* /dev/crypto memory charged to proc */
                                        /* protected by p_lock */
        clock_t p_ttime;                /* buffered task time */

        /*
         * The user structure
         */
        struct user p_user;             /* (see sys/user.h) */
} proc_t;
______unchanged_portion_omitted_
```

****

11325 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/sysinfo.h
6583 remove whole-process swapping

****

______***unchanged_portion_omitted_***

```
typedef struct cpu_vm_stats {
        uint64_t pgrec;                 /* page reclaims (includes pageout) */
        uint64_t pgfrec;                /* page reclaims from free list */
        uint64_t pgin;                  /* pageins */
        uint64_t pgpgin;                /* pages paged in */
        uint64_t pgout;                 /* pageouts */
        uint64_t pgpgout;               /* pages paged out */
        uint64_t swapin;                /* swapins */
        uint64_t pgswapin;              /* pages swapped in */
        uint64_t swapout;               /* swapouts */
        uint64_t pgswapout;             /* pages swapped out */
        uint64_t zfod;                  /* pages zero filled on demand */
        uint64_t dfree;                 /* pages freed by daemon or auto */
        uint64_t scan;                  /* pages examined by pageout daemon */
        uint64_t rev;                   /* revolutions of page daemon hand */
        uint64_t hat_fault;             /* minor page faults via hat_fault() */
        uint64_t as_fault;              /* minor page faults via as_fault() */
        uint64_t maj_fault;             /* major page faults */
        uint64_t cow_fault;             /* copy-on-write faults */
        uint64_t prot_fault;            /* protection faults */
        uint64_t softlock;              /* faults due to software locking req */
        uint64_t kernel_asflt;          /* as_fault()s in kernel addr space */
        uint64_t pgrrun;                /* times pager scheduled */
        uint64_t execpgin;              /* executable pages paged in */
        uint64_t execpgout;             /* executable pages paged out */
        uint64_t execfree;              /* executable pages freed */
        uint64_t anonpgin;              /* anon pages paged in */
        uint64_t anonpgout;             /* anon pages paged out */
        uint64_t anonfree;              /* anon pages freed */
        uint64_t fspgin;                /* fs pages paged in */
        uint64_t fspgout;               /* fs pages paged out */
        uint64_t fsfree;                /* fs pages free */
} cpu_vm_stats_t;
```

______***unchanged_portion_omitted_***

```
**********************************************************
   15545 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/systm.h
6583 remove whole-process swapping
**********************************************************
/*
 * 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 (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_SYSTM_H
#define _SYS_SYSTM_H

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/proc.h>
#include <sys/dditypes.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * The pc_t is the type of the kernel's program counter.  In general, a
 * pc_t is a uintptr_t -- except for a sparcv9 kernel, in which case all
 * instruction text is below 4G, and a pc_t is thus a uint32_t.
 */
#ifdef __sparcv9
typedef uint32_t pc_t;
#else
typedef uintptr_t pc_t;
#endif

/*
 * Random set of variables used by more than one routine.
 */

#if defined(_KERNEL) || defined(_FAKE_KERNEL)
#include <sys/types32.h>
#include <sys/varargs.h>
#include <sys/uadmin.h>
```

```
extern int hz;                  /* system clock rate */
extern struct vnode *rootdir;   /* pointer to vnode of root directory */
extern struct vnode *devicesdir;        /* pointer to /devices vnode */
extern int interrupts_unleashed;        /* set after the spl0() in main() */

extern char runin;              /* scheduling flag */
extern char runout;             /* scheduling flag */
extern char wake_sched;         /* causes clock to wake swapper on next tick */
extern char wake_sched_sec;     /* causes clock to wake swapper after a sec */

extern pgcnt_t  maxmem;         /* max available memory (pages) */
extern pgcnt_t  physmem;        /* physical memory (pages) on this CPU */
extern pfn_t    physmax;        /* highest numbered physical page present */
extern pgcnt_t  physinstalled;  /* physical pages including PROM/boot use */

extern caddr_t  s_text;         /* start of kernel text segment */
extern caddr_t  e_text;         /* end of kernel text segment */
extern caddr_t  s_data;         /* start of kernel text segment */
extern caddr_t  e_data;         /* end of kernel text segment */

extern pgcnt_t  availrmem;      /* Available resident (not swapable)    */
                                /* memory in pages.                     */
extern pgcnt_t  availrmem_initial;      /* initial value of availrmem   */
extern pgcnt_t  segspt_minfree; /* low water mark for availrmem in seg_spt */
extern pgcnt_t  freemem;        /* Current free memory.                 */

extern dev_t    rootdev;        /* device of the root */
extern struct vnode *rootvp;    /* vnode of root device */
extern boolean_t root_is_svm;           /* root is a mirrored device flag */
extern boolean_t root_is_ramdisk;       /* root is boot_archive ramdisk */
extern uint32_t  ramdisk_size;          /* (KB) set only for sparc netboots */
extern char *volatile panicstr; /* panic string pointer */
extern va_list  panicargs;      /* panic arguments */
extern volatile int quiesce_active;     /* quiesce(9E) is in progress */

extern int      rstchown;       /* 1 ==> restrictive chown(2) semantics */
extern int      klustsize;

extern int      abort_enable;   /* Platform input-device abort policy */

extern int      audit_active;   /* Solaris Auditing module state */

extern int      avenrun[];      /* array of load averages */

extern char *isa_list;          /* For sysinfo's isalist option */

extern int noexec_user_stack;           /* patchable via /etc/system */
extern int noexec_user_stack_log;       /* patchable via /etc/system */

/*
 * Use NFS client operations in the global zone only.  Under contract with
 * admin/install; do not change without coordinating with that consolidation.
 */
extern int nfs_global_client_only;

extern void report_stack_exec(proc_t *, caddr_t);

extern void startup(void);
extern void clkstart(void);
extern void post_startup(void);
extern void kern_setup1(void);
extern void ka_init(void);
extern void nodename_set(void);

/*
 * for tod fault detection
 */
enum tod_fault_type {
        TOD_REVERSED = 0,
        TOD_STALLED,
        TOD_JUMPED,
        TOD_RATECHANGED,
        TOD_RDONLY,
        TOD_NOFAULT
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   26129 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/thread.h
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

typedef struct _kthread *kthread_id_t;

struct turnstile;
struct panic_trap_info;
struct upimutex;
struct kproject;
struct on_trap_data;
struct waitq;
struct _kcpc_ctx;
struct _kcpc_set;

/* Definition for kernel thread identifier type */
typedef uint64_t kt_did_t;

typedef struct _kthread {
        struct _kthread *t_link; /* dispq, sleepq, and free queue link */

        caddr_t t_stk;          /* base of stack (kernel sp value to use) */
        void    (*t_startpc)(void);     /* PC where thread started */
        struct cpu *t_bound_cpu; /* cpu bound to, or NULL if not bound */
        short   t_affinitycnt;  /* nesting level of kernel affinity-setting */
        short   t_bind_cpu;     /* user-specified CPU binding (-1 if none) */
        ushort_t t_flag;                /* modified only by current thread */
        ushort_t t_proc_flag;   /* modified holding ttproc(t)->p_lock */
        ushort_t t_schedflag;   /* modified holding thread_lock(t) */
        volatile char t_preempt;        /* don't preempt thread if set */
        volatile char t_preempt_lk;
        uint_t  t_state;        /* thread state (protected by thread_lock) */
        pri_t   t_pri;          /* assigned thread priority */
        pri_t   t_epri;         /* inherited thread priority */
        pri_t   t_cpri;         /* thread scheduling class priority */
        char    t_writer;       /* sleeping in lwp_rwlock_lock(RW_WRITE_LOCK) */
        uchar_t t_bindflag;     /* CPU and pset binding type */
        label_t t_pcb;          /* pcb, save area when switching */
        lwpchan_t t_lwpchan;    /* reason for blocking */
#define t_wchan0        t_lwpchan.lc_wchan0
#define t_wchan         t_lwpchan.lc_wchan
        struct _sobj_ops *t_sobj_ops;
        id_t    t_cid;          /* scheduling class id */
        struct thread_ops *t_clfuncs;   /* scheduling class ops vector */
        void    *t_cldata;      /* per scheduling class specific data */
        ctxop_t *t_ctx;         /* thread context */
        uintptr_t t_lofault;    /* ret pc for failed page faults */
        label_t *t_onfault;     /* on_fault() setjmp buf */
        struct on_trap_data *t_ontrap;  /* on_trap() protection data */
        caddr_t t_swap;         /* the bottom of the stack, if from segkp */
        lock_t  t_lock;         /* used to resume() a thread */
        uint8_t t_lockstat;     /* set while thread is in lockstat code */
        uint8_t t_pil;          /* interrupt thread PIL */
        disp_lock_t     t_pi_lock;      /* lock protecting t_prioinv list */
        char    t_nomigrate;    /* do not migrate if set */
        struct cpu      *t_cpu; /* CPU that thread last ran on */
        struct cpu      *t_weakbound_cpu;       /* cpu weakly bound to */
        struct lgrp_ld  *t_lpl; /* load average for home lgroup */
        void            *t_lgrp_reserv[2];      /* reserved for future */
        struct _kthread *t_intr; /* interrupted (pinned) thread */
        uint64_t        t_intr_start;   /* timestamp when time slice began */
        kt_did_t        t_did;  /* thread id for kernel debuggers */
        caddr_t t_tnf_tpdp;     /* Trace facility data pointer */
```

```
        struct _kcpc_ctx *t_cpc_ctx;    /* performance counter context */
        struct _kcpc_set *t_cpc_set;    /* set this thread has bound */

        /*
         * non swappable part of the lwp state.
         */
        id_t            t_tid;          /* lwp's id */
        id_t            t_waitfor;      /* target lwp id in lwp_wait() */
        struct sigqueue *t_sigqueue;    /* queue of siginfo structs */
        k_sigset_t      t_sig;          /* signals pending to this process */
        k_sigset_t      t_extsig;       /* signals sent from another contract */
        k_sigset_t      t_hold;         /* hold signal bit mask */
        k_sigset_t      t_sigwait;      /* sigtimedwait/sigfd accepting these */
        struct  _kthread *t_forw;       /* process's forward thread link */
        struct  _kthread *t_back;       /* process's backward thread link */
        struct  _kthread *t_thlink;     /* tid (lwpid) lookup hash link */
        klwp_t  *t_lwp;                 /* thread's lwp pointer */
        struct  proc    *t_procp;       /* proc pointer */
        struct  t_audit_data *t_audit_data;     /* per thread audit data */
        struct  _kthread *t_next;       /* doubly linked list of all threads */
        struct  _kthread *t_prev;
        ushort_t t_whystop;             /* reason for stopping */
        ushort_t t_whatstop;            /* more detailed reason */
        int     t_dslot;                /* index in proc's thread directory */
        struct  pollstate *t_pollstate; /* state used during poll(2) */
        struct  pollcache *t_pollcache; /* to pass a pcache ptr by /dev/poll */
        struct  cred    *t_cred;        /* pointer to current cred */
        time_t  t_start;                /* start time, seconds since epoch */
        clock_t t_lbolt;                /* lbolt at last clock_tick() */
        hrtime_t t_stoptime;            /* timestamp at stop() */
        uint_t  t_pctcpu;               /* %cpu at last clock_tick(), binary */
                                        /* point at right of high-order bit */
        short   t_sysnum;               /* system call number */
        kcondvar_t      t_delay_cv;
        kmutex_t        t_delay_lock;

        /*
         * Pointer to the dispatcher lock protecting t_state and state-related
         * flags.  This pointer can change during waits on the lock, so
         * it should be grabbed only by thread_lock().
         */
        disp_lock_t     *t_lockp;       /* pointer to the dispatcher lock */
        ushort_t        t_oldspl;       /* spl level before dispatcher locked */
        volatile char   t_pre_sys;      /* pre-syscall work needed */
        lock_t          t_lock_flush;   /* for lock_mutex_flush() impl */
        struct _disp    *t_disp_queue;  /* run queue for chosen CPU */
        clock_t         t_disp_time;    /* last time this thread was running */
        uint_t          t_kpri_req;     /* kernel priority required */

        /*
         * Post-syscall / post-trap flags.
         *      No lock is required to set these.
         *      These must be cleared only by the thread itself.
         *
         *      t_astflag indicates that some post-trap processing is required,
         *              possibly a signal or a preemption.  The thread will not
         *              return to user with this set.
         *      t_post_sys indicates that some unusualy post-system call
         *              handling is required, such as an error or tracing.
         *      t_sig_check indicates that some condition in ISSIG() must be
         *              checked, but doesn't prevent returning to user.
         *      t_post_sys_ast is a way of checking whether any of these three
         *              flags are set.
         */
        union __tu {
                struct __ts {
                        volatile char   _t_astflag;     /* AST requested */
                        volatile char   _t_sig_check;   /* ISSIG required */
                        volatile char   _t_post_sys;    /* post_syscall req */
                        volatile char   _t_trapret;     /* call CL_TRAPRET */
                } _ts;
                volatile int    _t_post_sys_ast;        /* OR of these flags */
        } _tu;
#define t_astflag       _tu._ts._t_astflag
#define t_sig_check     _tu._ts._t_sig_check
#define t_post_sys      _tu._ts._t_post_sys
#define t_trapret       _tu._ts._t_trapret
#define t_post_sys_ast  _tu._t_post_sys_ast

        /*
         * Real time microstate profiling.
         */
                                        /* possible 4-byte filler */
        hrtime_t t_waitrq;              /* timestamp for run queue wait time */
        int     t_mstate;               /* current microstate */
        struct rprof {
                int     rp_anystate;            /* set if any state non-zero */
                uint_t  rp_state[NMSTATES];     /* mstate profiling counts */
        } *t_rprof;

        /*
         * There is a turnstile inserted into the list below for
         * every priority inverted synchronization object that
         * this thread holds.
         */

        struct turnstile *t_prioinv;

        /*
         * Pointer to the turnstile attached to the synchronization
         * object where this thread is blocked.
         */

        struct turnstile *t_ts;

        /*
         * kernel thread specific data
         *      Borrowed from userland implementation of POSIX tsd
         */
        struct tsd_thread {
                struct tsd_thread *ts_next;     /* threads with TSD */
                struct tsd_thread *ts_prev;     /* threads with TSD */
                uint_t            ts_nkeys;     /* entries in value array */
                void              **ts_value;   /* array of value/key */
        } *t_tsd;

        clock_t         t_stime;        /* time stamp used by the swapper */
        struct door_data *t_door;       /* door invocation data */
        kmutex_t        *t_plockp;      /* pointer to process’s p_lock */

        struct sc_shared *t_schedctl;   /* scheduler activations shared data */
        uintptr_t       t_sc_uaddr;     /* user-level address of shared data */

        struct cpupart  *t_cpupart;     /* partition containing thread */
        int             t_bind_pset;    /* processor set binding */

        struct copyops  *t_copyops;     /* copy in/out ops vector */

        caddr_t         t_stkbase;      /* base of the the stack */
        struct page     *t_red_pp;      /* if non-NULL, redzone is mapped */

        afd_t           t_activefd;     /* active file descriptor table */
```

```
        struct _kthread *t_priforw;     /* sleepq per-priority sublist */
        struct _kthread *t_priback;

        struct sleepq   *t_sleepq;      /* sleep queue thread is waiting on */
        struct panic_trap_info *t_panic_trap;   /* saved data from fatal trap */
        int             *t_lgrp_affinity;       /* lgroup affinity */
        struct upimutex *t_upimutex;    /* list of upimutexes owned by thread */
        uint32_t        t_nupinest;     /* number of nested held upi mutexes */
        struct kproject *t_proj;        /* project containing this thread */
        uint8_t         t_unpark;       /* modified holding t_delay_lock */
        uint8_t         t_release;      /* lwp_release() waked up the thread */
        uint8_t         t_hatdepth;     /* depth of recursive hat_memloads */
        uint8_t         t_xpvcntr;      /* see xen_block_migrate() */
        kcondvar_t      t_joincv;       /* cv used to wait for thread exit */
        void            *t_taskq;       /* for threads belonging to taskq */
        hrtime_t        t_anttime;      /* most recent time anticipatory load */
                                        /*      was added to an lgroup’s load */
                                        /*      on this thread’s behalf */
        char            *t_pdmsg;       /* privilege debugging message */

        uint_t          t_predcache;    /* DTrace predicate cache */
        hrtime_t        t_dtrace_vtime; /* DTrace virtual time */
        hrtime_t        t_dtrace_start; /* DTrace slice start time */

        uint8_t         t_dtrace_stop;  /* indicates a DTrace-desired stop */
        uint8_t         t_dtrace_sig;   /* signal sent via DTrace’s raise() */

        union __tdu {
                struct __tds {
                        uint8_t _t_dtrace_on;   /* hit a fasttrap tracepoint */
                        uint8_t _t_dtrace_step; /* about to return to kernel */
                        uint8_t _t_dtrace_ret;  /* handling a return probe */
                        uint8_t _t_dtrace_ast;  /* saved ast flag */
#ifdef __amd64
                        uint8_t _t_dtrace_reg;  /* modified register */
#endif
                } _tds;
                ulong_t _t_dtrace_ft;           /* bitwise or of these flags */
        } _tdu;
#define t_dtrace_ft     _tdu._t_dtrace_ft
#define t_dtrace_on     _tdu._tds._t_dtrace_on
#define t_dtrace_step   _tdu._tds._t_dtrace_step
#define t_dtrace_ret    _tdu._tds._t_dtrace_ret
#define t_dtrace_ast    _tdu._tds._t_dtrace_ast
#ifdef __amd64
#define t_dtrace_reg    _tdu._tds._t_dtrace_reg
#endif

        uintptr_t       t_dtrace_pc;    /* DTrace saved pc from fasttrap */
        uintptr_t       t_dtrace_npc;   /* DTrace next pc from fasttrap */
        uintptr_t       t_dtrace_scrpc; /* DTrace per-thread scratch location */
        uintptr_t       t_dtrace_astpc; /* DTrace return sequence location */
#ifdef __amd64
        uint64_t        t_dtrace_regv;  /* DTrace saved reg from fasttrap */
#endif
        hrtime_t        t_hrtime;       /* high-res last time on cpu */
        kmutex_t        t_ctx_lock;     /* protects t_ctx in removectx() */
        struct waitq    *t_waitq;       /* wait queue */
        kmutex_t        t_wait_mutex;   /* used in CV wait functions */
} kthread_t;

/*
 * Thread flag (t_flag) definitions.
 *      These flags must be changed only for the current thread,
 *      and not during preemption code, since the code being
 *      preempted could be modifying the flags.
 *
 *      For the most part these flags do not need locking.
 *      The following flags will only be changed while the thread_lock is held,
 *      to give assurrance that they are consistent with t_state:
 *              T_WAKEABLE
 */
#define T_INTR_THREAD   0x0001  /* thread is an interrupt thread */
#define T_WAKEABLE      0x0002  /* thread is blocked, signals enabled */
#define T_TOMASK        0x0004  /* use lwp_sigoldmask on return from signal */
#define T_TALLOCSTK     0x0008  /* thread structure allocated from stk */
#define T_FORKALL       0x0010  /* thread was cloned by forkall() */
#define T_WOULDBLOCK    0x0020  /* for lockfs */
#define T_DONTBLOCK     0x0040  /* for lockfs */
#define T_DONTPEND      0x0080  /* for lockfs */
#define T_SYS_PROF      0x0100  /* profiling on for duration of system call */
#define T_WAITCVSEM     0x0200  /* waiting for a lwp_cv or lwp_sema on sleepq */
#define T_WATCHPT       0x0400  /* thread undergoing a watchpoint emulation */
#define T_PANIC         0x0800  /* thread initiated a system panic */
#define T_LWPREUSE      0x1000  /* stack and LWP can be reused */
#define T_CAPTURING     0x2000  /* thread is in page capture logic */
#define T_VFPARENT      0x4000  /* thread is vfork parent, must call vfwait */
#define T_DONTDTRACE    0x8000  /* disable DTrace probes */

/*
 * Flags in t_proc_flag.
 *      These flags must be modified only when holding the p_lock
 *      for the associated process.
 */
#define TP_DAEMON       0x0001  /* this is an LWP_DAEMON lwp */
#define TP_HOLDLWP      0x0002  /* hold thread's lwp */
#define TP_TWAIT        0x0004  /* wait to be freed by lwp_wait() */
#define TP_LWPEXIT      0x0008  /* lwp has exited */
#define TP_PRSTOP       0x0010  /* thread is being stopped via /proc */
#define TP_CHKPT        0x0020  /* thread is being stopped via CPR checkpoint */
#define TP_EXITLWP      0x0040  /* terminate this lwp */
#define TP_PRVSTOP      0x0080  /* thread is virtually stopped via /proc */
#define TP_MSACCT       0x0100  /* collect micro-state accounting information */
#define TP_STOPPING     0x0200  /* thread is executing stop() */
#define TP_WATCHPT      0x0400  /* process has watchpoints in effect */
#define TP_PAUSE        0x0800  /* process is being stopped via pauselwps() */
#define TP_CHANGEBIND   0x1000  /* thread has a new cpu/cpupart binding */
#define TP_ZTHREAD      0x2000  /* this is a kernel thread for a zone */
#define TP_WATCHSTOP    0x4000  /* thread is stopping via holdwatch() */

/*
 * Thread scheduler flag (t_schedflag) definitions.
 *      The thread must be locked via thread_lock() or equiv. to change these.
 */
#define TS_LOAD         0x0001  /* thread is in memory */
#define TS_DONT_SWAP    0x0002  /* thread/lwp should not be swapped */
#define TS_SWAPENQ      0x0004  /* swap thread when it reaches a safe point */
#define TS_ON_SWAPQ     0x0008  /* thread is on the swap queue */
#define TS_SIGNALLED    0x0010  /* thread was awakened by cv_signal() */
#define TS_PROJWAITQ    0x0020  /* thread is on its project's waitq */
#define TS_ZONEWAITQ    0x0040  /* thread is on its zone's waitq */
#define TS_CSTART       0x0100  /* setrun() by continuelwps() */
#define TS_UNPAUSE      0x0200  /* setrun() by unpauselwps() */
#define TS_XSTART       0x0400  /* setrun() by SIGCONT */
#define TS_PSTART       0x0800  /* setrun() by /proc */
#define TS_RESUME       0x1000  /* setrun() by CPR resume process */
#define TS_CREATE       0x2000  /* setrun() by syslwp_create() */
#define TS_RUNQMATCH    0x4000  /* exact run queue balancing by setbackdq() */
#define TS_ALLSTART     \
        (TS_CSTART|TS_UNPAUSE|TS_XSTART|TS_PSTART|TS_RESUME|TS_CREATE)
#define TS_ANYWAITQ     (TS_PROJWAITQ|TS_ZONEWAITQ)
```

```
/*
 * Thread binding types
 */
#define TB_ALLHARD      0
#define TB_CPU_SOFT     0x01            /* soft binding to CPU */
#define TB_PSET_SOFT    0x02            /* soft binding to pset */

#define TB_CPU_SOFT_SET(t)              ((t)->t_bindflag |= TB_CPU_SOFT)
#define TB_CPU_HARD_SET(t)              ((t)->t_bindflag &= ~TB_CPU_SOFT)
#define TB_PSET_SOFT_SET(t)             ((t)->t_bindflag |= TB_PSET_SOFT)
#define TB_PSET_HARD_SET(t)             ((t)->t_bindflag &= ~TB_PSET_SOFT)
#define TB_CPU_IS_SOFT(t)               ((t)->t_bindflag & TB_CPU_SOFT)
#define TB_CPU_IS_HARD(t)               (!TB_CPU_IS_SOFT(t))
#define TB_PSET_IS_SOFT(t)              ((t)->t_bindflag & TB_PSET_SOFT)

/*
 * No locking needed for AST field.
 */
#define aston(t)                ((t)->t_astflag = 1)
#define astoff(t)               ((t)->t_astflag = 0)

/* True if thread is stopped on an event of interest */
#define ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
                        !((t)->t_schedflag & TS_PSTART))

/* True if thread is asleep and wakeable */
#define ISWAKEABLE(t) (((t)->t_state == TS_SLEEP && \
                        ((t)->t_flag & T_WAKEABLE)))

/* True if thread is on the wait queue */
#define ISWAITING(t) ((t)->t_state == TS_WAIT)

/* similar to ISTOPPED except the event of interest is CPR */
#define CPR_ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
                        !((t)->t_schedflag & TS_RESUME))

/*
 * True if thread is virtually stopped (is or was asleep in
 * one of the lwp_*() system calls and marked to stop by /proc.)
 */
#define VSTOPPED(t)     ((t)->t_proc_flag & TP_PRVSTOP)

/* similar to VSTOPPED except the point of interest is CPR */
#define CPR_VSTOPPED(t)                         \
        ((t)->t_state == TS_SLEEP &&            \
        (t)->t_wchan0 != NULL &&                \
        ((t)->t_flag & T_WAKEABLE) &&           \
        ((t)->t_proc_flag & TP_CHKPT))

/* True if thread has been stopped by hold*() or was created stopped */
#define SUSPENDED(t) ((t)->t_state == TS_STOPPED && \
        ((t)->t_schedflag & (TS_CSTART|TS_UNPAUSE)) != (TS_CSTART|TS_UNPAUSE))

/* True if thread possesses an inherited priority */
#define INHERITED(t)    ((t)->t_epri != 0)

/* The dispatch priority of a thread */
#define DISP_PRIO(t) ((t)->t_epri > (t)->t_pri ? (t)->t_epri : (t)->t_pri)

/* The assigned priority of a thread */
#define ASSIGNED_PRIO(t)        ((t)->t_pri)

/*
 * Macros to determine whether a thread can be swapped.
 * If t_lock is held, the thread is either on a processor or being swapped.
 */
#define SWAP_OK(t)      (!LOCK_HELD(&(t)->t_lock))

/*
 * proctot(x)
 *      convert a proc pointer to a thread pointer. this only works with
 *      procs that have only one lwp.
 *
 * proctolwp(x)
 *      convert a proc pointer to a lwp pointer. this only works with
 *      procs that have only one lwp.
 *
 * ttolwp(x)
 *      convert a thread pointer to its lwp pointer.
 *
 * ttoproc(x)
 *      convert a thread pointer to its proc pointer.
 *
 * ttoproj(x)
 *      convert a thread pointer to its project pointer.
 *
 * ttozone(x)
 *      convert a thread pointer to its zone pointer.
 *
 * lwptot(x)
 *      convert a lwp pointer to its thread pointer.
 *
 * lwptoproc(x)
 *      convert a lwp to its proc pointer.
 */
#define proctot(x)      ((x)->p_tlist)
#define proctolwp(x)    ((x)->p_tlist->t_lwp)
#define ttolwp(x)       ((x)->t_lwp)
#define ttoproc(x)      ((x)->t_procp)
#define ttoproj(x)      ((x)->t_proj)
#define ttozone(x)      ((x)->t_procp->p_zone)
#define lwptot(x)       ((x)->lwp_thread)
#define lwptoproc(x)    ((x)->lwp_procp)

#define t_pc            t_pcb.val[0]
#define t_sp            t_pcb.val[1]

#ifdef  _KERNEL

extern  kthread_t       *threadp(void); /* inline, returns thread pointer */
#define curthread       (threadp())             /* current thread pointer */
#define curproc         (ttoproc(curthread))    /* current process pointer */
#define curproj         (ttoproj(curthread))    /* current project pointer */
#define curzone         (curproc->p_zone)       /* current zone pointer */

extern  struct _kthread t0;             /* the scheduler thread */
extern  kmutex_t        pidlock;        /* global process lock */

/*
 * thread_free_lock is used by the tick accounting thread to keep a thread
 * from being freed while it is being examined.
 *
 * Thread structures are 32-byte aligned structures. That is why we use the
 * following formula.
 */
#define THREAD_FREE_BITS        10
#define THREAD_FREE_NUM         (1 << THREAD_FREE_BITS)
#define THREAD_FREE_MASK        (THREAD_FREE_NUM - 1)
#define THREAD_FREE_1           PTR24_LSB
#define THREAD_FREE_2           (PTR24_LSB + THREAD_FREE_BITS)
#define THREAD_FREE_SHIFT(t)    \
```

```
        (((ulong_t)(t) >> THREAD_FREE_1) ^ ((ulong_t)(t) >> THREAD_FREE_2))
#define THREAD_FREE_HASH(t)     (THREAD_FREE_SHIFT(t) & THREAD_FREE_MASK)

typedef struct thread_free_lock {
        kmutex_t        tf_lock;
        uchar_t         tf_pad[64 - sizeof (kmutex_t)];
} thread_free_lock_t;
```

______***unchanged_portion_omitted_***

```
/*
 * Macros to change thread state and the associated lock.
 */
#define THREAD_SET_STATE(tp, state, lp) \
                ((tp)->t_state = state, (tp)->t_lockp = lp)

/*
 * Point it at the transition lock, which is always held.
 * The previosly held lock is dropped.
 */
#define THREAD_TRANSITION(tp)   thread_transition(tp);
/*
 * Set the thread's lock to be the transition lock, without dropping
 * previosly held lock.
 */
#define THREAD_TRANSITION_NOLOCK(tp)    ((tp)->t_lockp = &transition_lock)

/*
 * Put thread in run state, and set the lock pointer to the dispatcher queue
 * lock pointer provided.  This lock should be held.
 */
#define THREAD_RUN(tp, lp)      THREAD_SET_STATE(tp, TS_RUN, lp)

/*
 * Put thread in wait state, and set the lock pointer to the wait queue
 * lock pointer provided.  This lock should be held.
 */
#define THREAD_WAIT(tp, lp)     THREAD_SET_STATE(tp, TS_WAIT, lp)

/*
 * Put thread in run state, and set the lock pointer to the dispatcher queue
 * lock pointer provided (i.e., the "swapped_lock").  This lock should be held.
 */
#define THREAD_SWAP(tp, lp)     THREAD_SET_STATE(tp, TS_RUN, lp)

/*
 * Put the thread in zombie state and set the lock pointer to NULL.
 * The NULL will catch anything that tries to lock a zombie.
 */
#define THREAD_ZOMB(tp)         THREAD_SET_STATE(tp, TS_ZOMB, NULL)

/*
 * Set the thread into ONPROC state, and point the lock at the CPUs
 * lock for the onproc thread(s).  This lock should be held, so the
 * thread deoes not become unlocked, since these stores can be reordered.
 */
#define THREAD_ONPROC(tp, cpu)  \
                THREAD_SET_STATE(tp, TS_ONPROC, &(cpu)->cpu_thread_lock)

/*
 * Set the thread into the TS_SLEEP state, and set the lock pointer to
 * to some sleep queue's lock.  The new lock should already be held.
 */
#define THREAD_SLEEP(tp, lp)    {                               \
                        disp_lock_t     *tlp;                   \
                        tlp = (tp)->t_lockp;                    \
                        THREAD_SET_STATE(tp, TS_SLEEP, lp);     \
                        disp_lock_exit_high(tlp);               \
                        }

/*
 * Interrupt threads are created in TS_FREE state, and their lock
 * points at the associated CPU's lock.
 */
#define THREAD_FREEINTR(tp, cpu)        \
                THREAD_SET_STATE(tp, TS_FREE, &(cpu)->cpu_thread_lock)

/* if tunable kmem_stackinfo is set, fill kthread stack with a pattern */
#define KMEM_STKINFO_PATTERN    0xbadcbadcbadcbadcULL

/*
 * If tunable kmem_stackinfo is set, log the latest KMEM_LOG_STK_USAGE_SIZE
 * dead kthreads that used their kernel stack the most.
 */
#define KMEM_STKINFO_LOG_SIZE   16

/* kthread name (cmd/lwpid) string size in the stackinfo log */
#define KMEM_STKINFO_STR_SIZE   64

/*
 * stackinfo logged data.
 */
typedef struct kmem_stkinfo {
        caddr_t kthread;        /* kthread pointer */
        caddr_t t_startpc;      /* where kthread started */
        caddr_t start;          /* kthread stack start address */
        size_t  stksz;          /* kthread stack size */
        size_t  percent;        /* kthread stack high water mark */
        id_t    t_tid;          /* kthread id */
        char    cmd[KMEM_STKINFO_STR_SIZE];     /* kthread name (cmd/lwpid) */
} kmem_stkinfo_t;
```
______***unchanged_portion_omitted_***

```
************************************************************
    5125 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/vmsystm.h
6583 remove whole-process swapping
************************************************************
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#ifndef _SYS_VMSYSTM_H
#define _SYS_VMSYSTM_H

#include <sys/proc.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Miscellaneous virtual memory subsystem variables and structures.
 */
#ifdef _KERNEL
extern pgcnt_t  freemem;        /* remaining blocks of free memory */
extern pgcnt_t  avefree;        /* 5 sec moving average of free memory */
extern pgcnt_t  avefree30;      /* 30 sec moving average of free memory */
extern pgcnt_t  deficit;        /* estimate of needs of new swapped in procs */
extern pgcnt_t  nscan;          /* number of scans in last second */
extern pgcnt_t  desscan;        /* desired pages scanned per second */
extern pgcnt_t  slowscan;
extern pgcnt_t  fastscan;
extern pgcnt_t  pushes;         /* number of pages pushed to swap device */
```

```
/* writable copies of tunables */
extern pgcnt_t  maxpgio;        /* max paging i/o per sec before start swaps */
extern pgcnt_t  lotsfree;       /* max free before clock freezes */
extern pgcnt_t  desfree;        /* minimum free pages before swapping begins */
extern pgcnt_t  minfree;        /* no of pages to try to keep free via daemon */
extern pgcnt_t  needfree;       /* no of pages currently being waited for */
extern pgcnt_t  throttlefree;   /* point at which we block PG_WAIT calls */
extern pgcnt_t  pageout_reserve; /* point at which we deny non-PG_WAIT calls */
extern pgcnt_t  pages_before_pager; /* XXX */

/*
 * TRUE if the pageout daemon, fsflush daemon or the scheduler.  These
 * processes can't sleep while trying to free up memory since a deadlock
 * will occur if they do sleep.
 */
#define NOMEMWAIT() (ttoproc(curthread) == proc_pageout || \
                        ttoproc(curthread) == proc_fsflush || \
                        ttoproc(curthread) == proc_sched)

/* insure non-zero */
#define nz(x)   ((x) != 0 ? (x) : 1)

/*
 * Flags passed by the swapper to swapout routines of each
 * scheduling class.
 */
#define HARDSWAP        1
#define SOFTSWAP        2

/*
 * Values returned by valid_usr_range()
 */
#define RANGE_OKAY      (0)
#define RANGE_BADADDR   (1)
#define RANGE_BADPROT   (2)

/*
 * map_pgsz: temporary - subject to change.
 */
#define MAPPGSZ_VA      0x01
#define MAPPGSZ_STK     0x02
#define MAPPGSZ_HEAP    0x04
#define MAPPGSZ_ISM     0x08

/*
 * Flags for map_pgszcvec
 */
#define MAPPGSZC_SHM    0x01
#define MAPPGSZC_PRIVM  0x02
#define MAPPGSZC_STACK  0x04
#define MAPPGSZC_HEAP   0x08

/*
 * vacalign values for choose_addr
 */
#define ADDR_NOVACALIGN 0
#define ADDR_VACALIGN   1

struct as;
struct page;
struct anon;

extern int maxslp;
extern ulong_t pginrate;
extern ulong_t pgoutrate;
extern void swapout_lwp(klwp_t *);
extern  int valid_va_range(caddr_t *basep, size_t *lenp, size_t minlen,
                int dir);
extern  int valid_va_range_aligned(caddr_t *basep, size_t *lenp,
    size_t minlen, int dir, size_t align, size_t redzone, size_t off);

extern  int valid_usr_range(caddr_t, size_t, uint_t, struct as *, caddr_t);
extern  int useracc(void *, size_t, int);
extern  size_t map_pgsz(int maptype, struct proc *p, caddr_t addr, size_t len,
    int memcntl);
extern  uint_t map_pgszcvec(caddr_t addr, size_t size, uintptr_t off, int flags,
    int type, int memcntl);
extern int choose_addr(struct as *as, caddr_t *addrp, size_t len, offset_t off,
    int vacalign, uint_t flags);
extern  void map_addr(caddr_t *addrp, size_t len, offset_t off, int vacalign,
    uint_t flags);
extern  int map_addr_vacalign_check(caddr_t, u_offset_t);
extern  void map_addr_proc(caddr_t *addrp, size_t len, offset_t off,
    int vacalign, caddr_t userlimit, struct proc *p, uint_t flags);
extern  void vmmeter(void);
extern  int cow_mapin(struct as *, caddr_t, caddr_t, struct page **,
        struct anon **, size_t *, int);

extern  caddr_t ppmapin(struct page *, uint_t, caddr_t);
extern  void    ppmapout(caddr_t);

extern  int pf_is_memory(pfn_t);

extern  void    dcache_flushall(void);

extern  void    *boot_virt_alloc(void *addr, size_t size);

extern  size_t  exec_get_spslew(void);

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

****
18138 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/vm/anon.h
6583 remove whole-process swapping
****
______***unchanged_portion_omitted_***

```
extern  struct k_anoninfo k_anoninfo;

extern void     anon_init(void);
extern struct   anon *anon_alloc(struct vnode *, anoff_t);
extern void     anon_dup(struct anon_hdr *, ulong_t,
                    struct anon_hdr *, ulong_t, size_t);
extern void     anon_dup_fill_holes(struct anon_hdr *, ulong_t,
                    struct anon_hdr *, ulong_t, size_t, uint_t, int);
extern int      anon_fill_cow_holes(struct seg *, caddr_t, struct anon_hdr *,
                    ulong_t, struct vnode *, u_offset_t, size_t, uint_t,
                    uint_t, struct vpage [], struct cred *);
extern void     anon_free(struct anon_hdr *, ulong_t, size_t);
extern void     anon_free_pages(struct anon_hdr *, ulong_t, size_t, uint_t);
extern void     anon_disclaim(struct anon_map *, ulong_t, size_t);
extern int      anon_getpage(struct anon **, uint_t *, struct page **,
                    size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
extern int      swap_getconpage(struct vnode *, u_offset_t, size_t,
                    uint_t *, page_t *[], size_t, page_t *, uint_t *,
                    spgcnt_t *, struct seg *, caddr_t,
                    enum seg_rw, struct cred *);
extern int      anon_map_getpages(struct anon_map *, ulong_t,
                    uint_t, struct seg *, caddr_t, uint_t,
                    uint_t *, page_t *[], uint_t *,
                    struct vpage [], enum seg_rw, int, int, int, struct cred *);
extern int      anon_map_privatepages(struct anon_map *, ulong_t,
                    uint_t, struct seg *, caddr_t, uint_t,
                    page_t *[], struct vpage [], int, int, struct cred *);
extern struct   page *anon_private(struct anon **, struct seg *,
                    caddr_t, uint_t, struct page *,
                    int, struct cred *);
extern struct   page *anon_zero(struct seg *, caddr_t,
                    struct anon **, struct cred *);
extern int      anon_map_createpages(struct anon_map *, ulong_t,
                    size_t, struct page **,
                    struct seg *, caddr_t,
                    enum seg_rw, struct cred *);
extern int      anon_map_demotepages(struct anon_map *, ulong_t,
                    struct seg *, caddr_t, uint_t,
                    struct vpage [], struct cred *);
extern void     anon_shmap_free_pages(struct anon_map *, ulong_t, size_t);
extern int      anon_resvmem(size_t, boolean_t, zone_t *, int);
extern void     anon_unresvmem(size_t, zone_t *);
extern struct   anon_map *anonmap_alloc(size_t, size_t, int);
extern void     anonmap_free(struct anon_map *);
extern void     anonmap_purge(struct anon_map *);
extern void     anon_swap_free(struct anon *, struct page *);
extern void     anon_decref(struct anon *);
extern int      non_anon(struct anon_hdr *, ulong_t, u_offset_t *, size_t *);
extern pgcnt_t  anon_pages(struct anon_hdr *, ulong_t, pgcnt_t);
extern int      anon_swap_adjust(pgcnt_t);
extern void     anon_swap_restore(pgcnt_t);
extern struct   anon_hdr *anon_create(pgcnt_t, int);
extern void     anon_release(struct anon_hdr *, pgcnt_t);
extern struct   anon *anon_get_ptr(struct anon_hdr *, ulong_t);
extern ulong_t  *anon_get_slot(struct anon_hdr *, ulong_t);
extern struct   anon *anon_get_next_ptr(struct anon_hdr *, ulong_t *);
extern int      anon_set_ptr(struct anon_hdr *, ulong_t, struct anon *, int);
extern int      anon_copy_ptr(struct anon_hdr *, ulong_t,
                    struct anon_hdr *, ulong_t, pgcnt_t, int);
```

```
extern pgcnt_t  anon_grow(struct anon_hdr *, ulong_t *, pgcnt_t, pgcnt_t, int);
extern void     anon_array_enter(struct anon_map *, ulong_t,
                        anon_sync_obj_t *);
extern int      anon_array_try_enter(struct anon_map *, ulong_t,
                        anon_sync_obj_t *);
extern void     anon_array_exit(anon_sync_obj_t *);

/*
 * anon_resv checks to see if there is enough swap space to fulfill a
 * request and if so, reserves the appropriate anonymous memory resources.
 * anon_checkspace just checks to see if there is space to fulfill the request,
 * without taking any resources.  Both return 1 if successful and 0 if not.
 *
 * Macros are provided as anon reservation is usually charged to the zone of
 * the current process.  In some cases (such as anon reserved by tmpfs), a
 * zone pointer is needed to charge the appropriate zone.
 */
#define anon_unresv(size)               anon_unresvmem(size, curproc->p_zone)
#define anon_unresv_zone(size, zone)    anon_unresvmem(size, zone)
#define anon_resv(size)                 \
        anon_resvmem((size), 1, curproc->p_zone, 1)
#define anon_resv_zone(size, zone)      anon_resvmem((size), 1, zone, 1)
#define anon_checkspace(size, zone)     anon_resvmem((size), 0, zone, 0)
#define anon_try_resv_zone(size, zone)  anon_resvmem((size), 1, zone, 0)

/*
 * Flags to anon_private
 */
#define STEAL_PAGE      0x1     /* page can be stolen */
#define LOCK_PAGE       0x2     /* page must be ‘‘logically’’ locked */

/*
 * SEGKP ANON pages that are locked are assumed to be LWP stack pages
 * and thus count towards the user pages locked count.
 * This value is protected by the same lock as availrmem.
 */
extern pgcnt_t anon_segkp_pages_locked;

extern int anon_debug;

#ifdef ANON_DEBUG

#define A_ANON  0x01
#define A_RESV  0x02
#define A_MRESV 0x04

/* vararg-like debugging macro. */
#define ANON_PRINT(f, printf_args) \
                if (anon_debug & f) \
                        printf printf_args

#else   /* ANON_DEBUG */

#define ANON_PRINT(f, printf_args)

#endif  /* ANON_DEBUG */

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   11482 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/vm/as.h
6583 remove whole-process swapping
**********************************************************
______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, \
        (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, \
        (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)

/*
 * 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_
```

```
**********************************************************
   19654 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/hat.h
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

  81 typedef void *hat_region_cookie_t;

  83 #ifdef  _KERNEL

  85 /*
  86  * One time hat initialization
  87  */
  88 void    hat_init(void);

  90 /*
  91  * Notify hat of a system dump
  92  */
  93 void    hat_dump(void);

  95 /*
  96  * Operations on an address space:
  97  *
  98  * struct hat *hat_alloc(as)
  99  *      allocated a hat structure for as.
 100  *
 101  * void hat_free_start(hat)
 102  *      informs hat layer process has finished executing but as has not
 103  *      been cleaned up yet.
 104  *
 105  * void hat_free_end(hat)
 106  *      informs hat layer as is being destroyed.  hat layer cannot use as
 107  *      pointer after this call.
 108  *
 109  * void hat_swapin(hat)
 110  *      allocate any hat resources required for process being swapped in.
 111  *
 112  * void hat_swapout(hat)
 113  *      deallocate hat resources for process being swapped out.
 114  *
 109  * size_t hat_get_mapped_size(hat)
 110  *      returns number of bytes that have valid mappings in hat.
 111  *
 112  * void hat_stats_enable(hat)
 113  * void hat_stats_disable(hat)
 114  *      enables/disables collection of stats for hat.
 115  *
 116  * int hat_dup(parenthat, childhat, addr, len, flags)
 117  *      Duplicate address translations of the parent to the child.  Supports
 118  *      the entire address range or a range depending on flag,
 119  *      zero returned on success, non-zero on error
 120  *
 121  * void hat_thread_exit(thread)
 122  *      Notifies the HAT that a thread is exiting, called after it has been
 123  *      reassigned to the kernel AS.
 124  */

 126 struct hat *hat_alloc(struct as *);
 127 void    hat_free_start(struct hat *);
 128 void    hat_free_end(struct hat *);
 129 int     hat_dup(struct hat *, struct hat *, caddr_t, size_t, uint_t);
 136 void    hat_swapin(struct hat *);
 137 void    hat_swapout(struct hat *);
 130 size_t  hat_get_mapped_size(struct hat *);
 131 int     hat_stats_enable(struct hat *);
```

```
 132 void    hat_stats_disable(struct hat *);
 133 void    hat_thread_exit(kthread_t *);

 135 /*
 136  * Operations on a named address within a segment:
 137  *
 138  * void hat_memload(hat, addr, pp, attr, flags)
 139  *      load/lock the given page struct
 140  *
 141  * void hat_memload_array(hat, addr, len, ppa, attr, flags)
 142  *      load/lock the given array of page structs
 143  *
 144  * void hat_devload(hat, addr, len, pf, attr, flags)
 145  *      load/lock the given page frame number
 146  *
 147  * void hat_unlock(hat, addr, len)
 148  *      unlock a given range of addresses
 149  *
 150  * void hat_unload(hat, addr, len, flags)
 151  * void hat_unload_callback(hat, addr, len, flags, callback)
 152  *      unload a given range of addresses (has optional callback)
 153  *
 154  * void hat_sync(hat, addr, len, flags)
 155  *      synchronize mapping with software data structures
 156  *
 157  * void hat_map(hat, addr, len, flags)
 158  *
 159  * void hat_setattr(hat, addr, len, attr)
 160  * void hat_clrattr(hat, addr, len, attr)
 161  * void hat_chgattr(hat, addr, len, attr)
 162  *      modify attributes for a range of addresses. skips any invalid mappings
 163  *
 164  * uint_t hat_getattr(hat, addr, *attr)
 165  *      returns attr for <hat,addr> in *attr.  returns 0 if there was a
 166  *      mapping and *attr is valid, nonzero if there was no mapping and
 167  *      *attr is not valid.
 168  *
 169  * size_t hat_getpagesize(hat, addr)
 170  *      returns pagesize in bytes for <hat, addr>. returns -1 if there is
 171  *      no mapping. This is an advisory call.
 172  *
 173  * pfn_t hat_getpfnum(hat, addr)
 174  *      returns pfn for <hat, addr> or PFN_INVALID if mapping is invalid.
 175  *
 176  * int hat_probe(hat, addr)
 177  *      return 0 if no valid mapping is present.  Faster version
 178  *      of hat_getattr in certain architectures.
 179  *
 180  * int hat_share(dhat, daddr, shat, saddr, len, szc)
 181  *
 182  * void hat_unshare(hat, addr, len, szc)
 183  *
 184  * void hat_chgprot(hat, addr, len, vprot)
 185  *      This is a deprecated call.  New segment drivers should store
 186  *      all attributes and use hat_*attr calls.
 187  *      Change the protections in the virtual address range
 188  *      given to the specified virtual protection.  If vprot is ~PROT_WRITE,
 189  *      then remove write permission, leaving the other permissions
 190  *      unchanged.  If vprot is ~PROT_USER, remove user permissions.
 191  *
 192  * void hat_flush_range(hat, addr, size)
 193  *      Invalidate a virtual address translation for the local CPU.
 194  */

 196 void    hat_memload(struct hat *, caddr_t, struct page *, uint_t, uint_t);
 197 void    hat_memload_array(struct hat *, caddr_t, size_t, struct page **,
                uint_t, uint_t);
void    hat_memload_region(struct hat *, caddr_t, struct page *, uint_t,
                uint_t, hat_region_cookie_t);
void    hat_memload_array_region(struct hat *, caddr_t, size_t, struct page **,
                uint_t, uint_t, hat_region_cookie_t);

void    hat_devload(struct hat *, caddr_t, size_t, pfn_t, uint_t, int);

void    hat_unlock(struct hat *, caddr_t, size_t);
void    hat_unlock_region(struct hat *, caddr_t, size_t, hat_region_cookie_t);

void    hat_unload(struct hat *, caddr_t, size_t, uint_t);
void    hat_unload_callback(struct hat *, caddr_t, size_t, uint_t,
                hat_callback_t *);
void    hat_flush_range(struct hat *, caddr_t, size_t);
void    hat_sync(struct hat *, caddr_t, size_t, uint_t);
void    hat_map(struct hat *, caddr_t, size_t, uint_t);
void    hat_setattr(struct hat *, caddr_t, size_t, uint_t);
void    hat_clrattr(struct hat *, caddr_t, size_t, uint_t);
void    hat_chgattr(struct hat *, caddr_t, size_t, uint_t);
uint_t  hat_getattr(struct hat *, caddr_t, uint_t *);
ssize_t hat_getpagesize(struct hat *, caddr_t);
pfn_t   hat_getpfnum(struct hat *, caddr_t);
int     hat_probe(struct hat *, caddr_t);
int     hat_share(struct hat *, caddr_t, struct hat *, caddr_t, size_t, uint_t);
void    hat_unshare(struct hat *, caddr_t, size_t, uint_t);
void    hat_chgprot(struct hat *, caddr_t, size_t, uint_t);
void    hat_reserve(struct as *, caddr_t, size_t);
pfn_t   va_to_pfn(void *);
uint64_t va_to_pa(void *);

/*
 * Kernel Physical Mapping (segkpm) hat interface routines.
 */
caddr_t hat_kpm_mapin(struct page *, struct kpme *);
void    hat_kpm_mapout(struct page *, struct kpme *, caddr_t);
caddr_t hat_kpm_mapin_pfn(pfn_t);
void    hat_kpm_mapout_pfn(pfn_t);
caddr_t hat_kpm_page2va(struct page *, int);
struct page *hat_kpm_vaddr2page(caddr_t);
int     hat_kpm_fault(struct hat *, caddr_t);
void    hat_kpm_mseghash_clear(int);
void    hat_kpm_mseghash_update(pgcnt_t, struct memseg *);
void    hat_kpm_addmem_mseg_update(struct memseg *, pgcnt_t, offset_t);
void    hat_kpm_addmem_mseg_insert(struct memseg *);
void    hat_kpm_addmem_memsegs_update(struct memseg *);
caddr_t hat_kpm_mseg_reuse(struct memseg *);
void    hat_kpm_delmem_mseg_update(struct memseg *, struct memseg **);
void    hat_kpm_split_mseg_update(struct memseg *, struct memseg **,
                        struct memseg *, struct memseg *, struct memseg *);
void    hat_kpm_walk(void (*)(void *, void *, size_t), void *);

/*
 * Operations on all translations for a given page(s)
 *
 * void hat_page_setattr(pp, flag)
 * void hat_page_clrattr(pp, flag)
 *      used to set/clr red/mod bits.
 *
 * uint hat_page_getattr(pp, flag)
 *      If flag is specified, returns 0 if attribute is disabled
 *      and non zero if enabled.  If flag specifes multiple attributs
 *      then returns 0 if ALL atriibutes are disabled.  This is an advisory
 *      call.
 *
 * int hat_pageunload(pp, forceflag)
```

```
 *      unload all translations attached to pp.
 *
 * uint_t hat_pagesync(pp, flags)
 *      get hw stats from hardware into page struct and reset hw stats
 *      returns attributes of page
 *
 * ulong_t hat_page_getshare(pp)
 *      returns approx number of mappings to this pp.  A return of 0 implies
 *      there are no mappings to the page.
 *
 * faultcode_t hat_softlock(hat, addr, lenp, ppp, flags);
 *      called to softlock pages for zero copy tcp
 *
 * void hat_page_demote(pp);
 *      unload all large mappings to pp and decrease p_szc of all
 *      constituent pages according to the remaining mappings.
 */

void    hat_page_setattr(struct page *, uint_t);
void    hat_page_clrattr(struct page *, uint_t);
uint_t  hat_page_getattr(struct page *, uint_t);
int     hat_pageunload(struct page *, uint_t);
uint_t  hat_pagesync(struct page *, uint_t);
ulong_t hat_page_getshare(struct page *);
int     hat_page_checkshare(struct page *, ulong_t);
faultcode_t hat_softlock(struct hat *, caddr_t, size_t *,
                        struct page **, uint_t);
void    hat_page_demote(struct page *);

/*
 * Rountine to expose supported HAT features to PIM.
 */
enum hat_features {
        HAT_SHARED_PT,          /* Shared page tables */
        HAT_DYNAMIC_ISM_UNMAP,  /* hat_pageunload() handles ISM pages */
        HAT_VMODSORT,           /* support for VMODSORT flag of vnode */
        HAT_SHARED_REGIONS      /* shared regions support */
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   10402 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg.h
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#define S_PURGE         (0x01)          /* seg should be purged in as_gap() */

struct  seg_ops {
        int     (*dup)(struct seg *, struct seg *);
        int     (*unmap)(struct seg *, caddr_t, size_t);
        void    (*free)(struct seg *);
        faultcode_t (*fault)(struct hat *, struct seg *, caddr_t, size_t,
            enum fault_type, enum seg_rw);
        faultcode_t (*faulta)(struct seg *, caddr_t);
        int     (*setprot)(struct seg *, caddr_t, size_t, uint_t);
        int     (*checkprot)(struct seg *, caddr_t, size_t, uint_t);
        int     (*kluster)(struct seg *, caddr_t, ssize_t);
        size_t  (*swapout)(struct seg *);
        int     (*sync)(struct seg *, caddr_t, size_t, int, uint_t);
        size_t  (*incore)(struct seg *, caddr_t, size_t, char *);
        int     (*lockop)(struct seg *, caddr_t, size_t, int, int, ulong_t *,
                        size_t);
        int     (*getprot)(struct seg *, caddr_t, size_t, uint_t *);
        u_offset_t      (*getoffset)(struct seg *, caddr_t);
        int     (*gettype)(struct seg *, caddr_t);
        int     (*getvp)(struct seg *, caddr_t, struct vnode **);
        int     (*advise)(struct seg *, caddr_t, size_t, uint_t);
        void    (*dump)(struct seg *);
        int     (*pagelock)(struct seg *, caddr_t, size_t, struct page ***,
                        enum lock_type, enum seg_rw);
        int     (*setpagesize)(struct seg *, caddr_t, size_t, uint_t);
        int     (*getmemid)(struct seg *, caddr_t, memid_t *);
        struct lgrp_mem_policy_info     *(*getpolicy)(struct seg *, caddr_t);
        int     (*capable)(struct seg *, segcapability_t);
        int     (*inherit)(struct seg *, caddr_t, size_t, uint_t);
};

#ifdef _KERNEL

/*
 * Generic segment operations
 */
extern  void    seg_init(void);
extern  struct  seg *seg_alloc(struct as *as, caddr_t base, size_t size);
extern  int     seg_attach(struct as *as, caddr_t base, size_t size,
                        struct seg *seg);
extern  void    seg_unmap(struct seg *seg);
extern  void    seg_free(struct seg *seg);

/*
 * functions for pagelock cache support
 */
typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
    struct page **, enum seg_rw, int);

extern  struct  page **seg_plookup(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);
extern  void    seg_pinactive(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,
    uint_t flags, seg_preclaim_cbfunc_t callback);

extern  void    seg_ppurge(struct seg *seg, struct anon_map *amp,
    uint_t flags);
extern  void    seg_ppurge_wiredpp(struct page **pp);
```

```
extern  int     seg_pinsert_check(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, uint_t flags);
extern  int     seg_pinsert(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,
    uint_t flags, seg_preclaim_cbfunc_t callback);

extern  void    seg_pasync_thread(void);
extern  void    seg_preap(void);
extern  int     seg_p_disable(void);
extern  void    seg_p_enable(void);

extern  segadvstat_t    segadvstat;

/*
 * Flags for pagelock cache support.
 * Flags argument is passed as uint_t to pcache routines.  upper 16 bits of
 * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
 * is set.
 */
#define SEGP_FORCE_WIRED        0x1     /* skip check against seg_pwindow */
#define SEGP_AMP                0x2     /* anon map's pcache entry */
#define SEGP_PSHIFT             0x4     /* addr pgsz shift for hash function */

/*
 * Return values for seg_pinsert and seg_pinsert_check functions.
 */
#define SEGP_SUCCESS            0       /* seg_pinsert() succeeded */
#define SEGP_FAIL               1       /* seg_pinsert() failed */

/* Page status bits for segop_incore */
#define SEG_PAGE_INCORE         0x01    /* VA has a page backing it */
#define SEG_PAGE_LOCKED         0x02    /* VA has a page that is locked */
#define SEG_PAGE_HASCOW         0x04    /* VA has a page with a copy-on-write */
#define SEG_PAGE_SOFTLOCK       0x08    /* VA has a page with softlock held */
#define SEG_PAGE_VNODEBACKED    0x10    /* Segment is backed by a vnode */
#define SEG_PAGE_ANON           0x20    /* VA has an anonymous page */
#define SEG_PAGE_VNODE          0x40    /* VA has a vnode page backing it */

#define SEGOP_DUP(s, n)             (*(s)->s_ops->dup)((s), (n))
#define SEGOP_UNMAP(s, a, l)        (*(s)->s_ops->unmap)((s), (a), (l))
#define SEGOP_FREE(s)               (*(s)->s_ops->free)((s))
#define SEGOP_FAULT(h, s, a, l, t, rw) \
                (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
#define SEGOP_FAULTA(s, a)          (*(s)->s_ops->faulta)((s), (a))
#define SEGOP_SETPROT(s, a, l, p)   (*(s)->s_ops->setprot)((s), (a), (l), (p))
#define SEGOP_CHECKPROT(s, a, l, p) (*(s)->s_ops->checkprot)((s), (a), (l), (p))
#define SEGOP_KLUSTER(s, a, d)      (*(s)->s_ops->kluster)((s), (a), (d))
#define SEGOP_SWAPOUT(s)            (*(s)->s_ops->swapout)((s))
#define SEGOP_SYNC(s, a, l, atr, f) \
                (*(s)->s_ops->sync)((s), (a), (l), (atr), (f))
#define SEGOP_INCORE(s, a, l, v)    (*(s)->s_ops->incore)((s), (a), (l), (v))
#define SEGOP_LOCKOP(s, a, l, atr, op, b, p) \
                (*(s)->s_ops->lockop)((s), (a), (l), (atr), (op), (b), (p))
#define SEGOP_GETPROT(s, a, l, p)   (*(s)->s_ops->getprot)((s), (a), (l), (p))
#define SEGOP_GETOFFSET(s, a)       (*(s)->s_ops->getoffset)((s), (a))
#define SEGOP_GETTYPE(s, a)         (*(s)->s_ops->gettype)((s), (a))
#define SEGOP_GETVP(s, a, vpp)      (*(s)->s_ops->getvp)((s), (a), (vpp))
#define SEGOP_ADVISE(s, a, l, b)    (*(s)->s_ops->advise)((s), (a), (l), (b))
#define SEGOP_DUMP(s)               (*(s)->s_ops->dump)((s))
#define SEGOP_PAGELOCK(s, a, l, p, t, rw) \
                (*(s)->s_ops->pagelock)((s), (a), (l), (p), (t), (rw))
#define SEGOP_SETPAGESIZE(s, a, l, szc) \
                (*(s)->s_ops->setpagesize)((s), (a), (l), (szc))
#define SEGOP_GETMEMID(s, a, mp)    (*(s)->s_ops->getmemid)((s), (a), (mp))
#define SEGOP_GETPOLICY(s, a)       (*(s)->s_ops->getpolicy)((s), (a))
```

```
#define SEGOP_CAPABLE(s, c)         (*(s)->s_ops->capable)((s), (c))
#define SEGOP_INHERIT(s, a, l, b)   (*(s)->s_ops->inherit)((s), (a), (l), (b))

#define seg_page(seg, addr) \
        (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)

#define seg_pages(seg) \
        (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)

#define IE_NOMEM        -1      /* internal to seg layer */
#define IE_RETRY        -2      /* internal to seg layer */
#define IE_REATTACH     -3      /* internal to seg layer */

/* Values for SEGOP_INHERIT */
#define SEGP_INH_ZERO   0x01

int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);

/* Delay/retry factors for seg_p_mem_config_pre_del */
#define SEGP_PREDEL_DELAY_FACTOR        4
/*
 * As a workaround to being unable to purge the pagelock
 * cache during a DR delete memory operation, we use
 * a stall threshold that is twice the maximum seen
 * during testing.  This workaround will be removed
 * when a suitable fix is found.
 */
#define SEGP_STALL_SECONDS      25
#define SEGP_STALL_THRESHOLD \
        (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

#ifdef VMDEBUG

uint_t  seg_page(struct seg *, caddr_t);
uint_t  seg_pages(struct seg *);

#endif  /* VMDEBUG */

boolean_t       seg_can_change_zones(struct seg *);
size_t          seg_swresv(struct seg *);

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
  113563 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_dev.c
6583 remove whole-process swapping
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - segment of a mapped device.
 *
 * This segment driver is used when mapping character special devices.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/systm.h>
#include <sys/vmsystm.h>
#include <sys/mman.h>
#include <sys/errno.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vnode.h>
#include <sys/proc.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/ddidevmap.h>
#include <sys/ddi_implfuncs.h>
```

```
#include <sys/lgrp.h>

#include <vm/page.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_dev.h>
#include <vm/seg_kp.h>
#include <vm/seg_kmem.h>
#include <vm/vpage.h>

#include <sys/sunddi.h>
#include <sys/esunddi.h>
#include <sys/fs/snode.h>


#if DEBUG
int segdev_debug;
#define DEBUGF(level, args) { if (segdev_debug >= (level)) cmn_err args; }
#else
#define DEBUGF(level, args)
#endif

/* Default timeout for devmap context management */
#define CTX_TIMEOUT_VALUE 0

#define HOLD_DHP_LOCK(dhp)  if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
                        { mutex_enter(&dhp->dh_lock); }

#define RELE_DHP_LOCK(dhp) if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
                        { mutex_exit(&dhp->dh_lock); }

#define round_down_p2(a, s)     ((a) & ~((s) - 1))
#define round_up_p2(a, s)       (((a) + (s) - 1) & ~((s) - 1))

/*
 * VA_PA_ALIGNED checks to see if both VA and PA are on pgsize boundary
 * VA_PA_PGSIZE_ALIGNED check to see if VA is aligned with PA w.r.t. pgsize
 */
#define VA_PA_ALIGNED(uvaddr, paddr, pgsize)            \
        (((uvaddr | paddr) & (pgsize - 1)) == 0)
#define VA_PA_PGSIZE_ALIGNED(uvaddr, paddr, pgsize)     \
        (((uvaddr ^ paddr) & (pgsize - 1)) == 0)

#define vpgtob(n)       ((n) * sizeof (struct vpage))   /* For brevity */

#define VTOCVP(vp)      (VTOS(vp)->s_commonvp)  /* we "know" it's an snode */

static struct devmap_ctx *devmapctx_list = NULL;
static struct devmap_softlock *devmap_slist = NULL;

/*
 * mutex, vnode and page for the page of zeros we use for the trash mappings.
 * One trash page is allocated on the first ddi_umem_setup call that uses it
 * XXX Eventually, we may want to combine this with what segnf does when all
 * hat layers implement HAT_NOFAULT.
 *
 * The trash page is used when the backing store for a userland mapping is
 * removed but the application semantics do not take kindly to a SIGBUS.
 * In that scenario, the applications pages are mapped to some dummy page
 * which returns garbage on read and writes go into a common place.
 * (Perfect for NO_FAULT semantics)
 * The device driver is responsible to communicating to the app with some
 * other mechanism that such remapping has happened and the app should take
 * corrective action.
 * We can also use an anonymous memory page as there is no requirement to
```

```c
 * keep the page locked, however this complicates the fault code. RFE.
 */
static struct vnode trashvp;
static struct page *trashpp;

/* Non-pageable kernel memory is allocated from the umem_np_arena. */
static vmem_t *umem_np_arena;

/* Set the cookie to a value we know will never be a valid umem_cookie */
#define DEVMAP_DEVMEM_COOKIE    ((ddi_umem_cookie_t)0x1)

/*
 * Macros to check if type of devmap handle
 */
#define cookie_is_devmem(c)     \
        ((c) == (struct ddi_umem_cookie *)DEVMAP_DEVMEM_COOKIE)

#define cookie_is_pmem(c)       \
        ((c) == (struct ddi_umem_cookie *)DEVMAP_PMEM_COOKIE)

#define cookie_is_kpmem(c)      (!cookie_is_devmem(c) && !cookie_is_pmem(c) &&\
        ((c)->type == KMEM_PAGEABLE))

#define dhp_is_devmem(dhp)      \
        (cookie_is_devmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

#define dhp_is_pmem(dhp)        \
        (cookie_is_pmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

#define dhp_is_kpmem(dhp)       \
        (cookie_is_kpmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

/*
 * Private seg op routines.
 */
static int      segdev_dup(struct seg *, struct seg *);
static int      segdev_unmap(struct seg *, caddr_t, size_t);
static void     segdev_free(struct seg *);
static faultcode_t segdev_fault(struct hat *, struct seg *, caddr_t, size_t,
                    enum fault_type, enum seg_rw);
static faultcode_t segdev_faulta(struct seg *, caddr_t);
static int      segdev_setprot(struct seg *, caddr_t, size_t, uint_t);
static int      segdev_checkprot(struct seg *, caddr_t, size_t, uint_t);
static void     segdev_badop(void);
static int      segdev_sync(struct seg *, caddr_t, size_t, int, uint_t);
static size_t   segdev_incore(struct seg *, caddr_t, size_t, char *);
static int      segdev_lockop(struct seg *, caddr_t, size_t, int, int,
                    ulong_t *, size_t);
static int      segdev_getprot(struct seg *, caddr_t, size_t, uint_t *);
static u_offset_t       segdev_getoffset(struct seg *, caddr_t);
static int      segdev_gettype(struct seg *, caddr_t);
static int      segdev_getvp(struct seg *, caddr_t, struct vnode **);
static int      segdev_advise(struct seg *, caddr_t, size_t, uint_t);
static void     segdev_dump(struct seg *);
static int      segdev_pagelock(struct seg *, caddr_t, size_t,
                    struct page ***, enum lock_type, enum seg_rw);
static int      segdev_setpagesize(struct seg *, caddr_t, size_t, uint_t);
static int      segdev_getmemid(struct seg *, caddr_t, memid_t *);
static lgrp_mem_policy_info_t   *segdev_getpolicy(struct seg *, caddr_t);
static int      segdev_capable(struct seg *, segcapability_t);

/*
 * XXX  this struct is used by rootnex_map_fault to identify
 *      the segment it has been passed. So if you make it
 *      "static" you'll need to fix rootnex_map_fault.
 */
```

```c
struct seg_ops segdev_ops = {
        segdev_dup,
        segdev_unmap,
        segdev_free,
        segdev_fault,
        segdev_faulta,
        segdev_setprot,
        segdev_checkprot,
        (int (*)())segdev_badop,        /* kluster */
        (size_t (*)(struct seg *))NULL, /* swapout */
        segdev_sync,                    /* sync */
        segdev_incore,
        segdev_lockop,                  /* lockop */
        segdev_getprot,
        segdev_getoffset,
        segdev_gettype,
        segdev_getvp,
        segdev_advise,
        segdev_dump,
        segdev_pagelock,
        segdev_setpagesize,
        segdev_getmemid,
        segdev_getpolicy,
        segdev_capable,
        seg_inherit_notsup
};
```
______***unchanged_portion_omitted_***

****
   45398 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kmem.c
6583 remove whole-process swapping
****
______*unchanged_portion_omitted_*

```
static struct seg_ops segkmem_ops = {
        SEGKMEM_BADOP(int),             /* dup */
        SEGKMEM_BADOP(int),             /* unmap */
        SEGKMEM_BADOP(void),            /* free */
        segkmem_fault,
        SEGKMEM_BADOP(faultcode_t),     /* faulta */
        segkmem_setprot,
        segkmem_checkprot,
        segkmem_kluster,
        SEGKMEM_BADOP(size_t),          /* swapout */
        SEGKMEM_BADOP(int),             /* sync */
        SEGKMEM_BADOP(size_t),          /* incore */
        SEGKMEM_BADOP(int),             /* lockop */
        SEGKMEM_BADOP(int),             /* getprot */
        SEGKMEM_BADOP(u_offset_t),      /* getoffset */
        SEGKMEM_BADOP(int),             /* gettype */
        SEGKMEM_BADOP(int),             /* getvp */
        SEGKMEM_BADOP(int),             /* advise */
        segkmem_dump,
        segkmem_pagelock,
        SEGKMEM_BADOP(int),             /* setpgsz */
        segkmem_getmemid,
        segkmem_getpolicy,              /* getpolicy */
        segkmem_capable,                /* capable */
        seg_inherit_notsup              /* inherit */
};
```
______*unchanged_portion_omitted_*

```
**********************************************************
   36713 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kp.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*      All Rights Reserved   */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * segkp is a segment driver that administers the allocation and deallocation
 * of pageable variable size chunks of kernel virtual address space. Each
 * allocated resource is page-aligned.
 *
 * The user may specify whether the resource should be initialized to 0,
 * include a redzone, or locked in memory.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/thread.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/buf.h>
#include <sys/mman.h>
#include <sys/vnode.h>
#include <sys/cmn_err.h>
#include <sys/swap.h>
#include <sys/tuneable.h>
#include <sys/kmem.h>
#include <sys/vmem.h>
#include <sys/cred.h>
#include <sys/dumphdr.h>
#include <sys/debug.h>
#include <sys/vtrace.h>
#include <sys/stack.h>
```

```
#include <sys/atomic.h>
#include <sys/archsystm.h>
#include <sys/lgrp.h>

#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_kp.h>
#include <vm/seg_kmem.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/hat.h>
#include <sys/bitmap.h>

/*
 * Private seg op routines
 */
static void     segkp_badop(void);
static void     segkp_dump(struct seg *seg);
static int      segkp_checkprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t prot);
static int      segkp_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
static int      segkp_pagelock(struct seg *seg, caddr_t addr, size_t len,
                        struct page ***page, enum lock_type type,
                        enum seg_rw rw);
static void     segkp_insert(struct seg *seg, struct segkp_data *kpd);
static void     segkp_delete(struct seg *seg, struct segkp_data *kpd);
static caddr_t  segkp_get_internal(struct seg *seg, size_t len, uint_t flags,
                        struct segkp_data **tkpd, struct anon_map *amp);
static void     segkp_release_internal(struct seg *seg,
                        struct segkp_data *kpd, size_t len);
static int      segkp_unlock(struct hat *hat, struct seg *seg, caddr_t vaddr,
                        size_t len, struct segkp_data *kpd, uint_t flags);
static int      segkp_load(struct hat *hat, struct seg *seg, caddr_t vaddr,
                        size_t len, struct segkp_data *kpd, uint_t flags);
static struct   segkp_data *segkp_find(struct seg *seg, caddr_t vaddr);
static int      segkp_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
static lgrp_mem_policy_info_t   *segkp_getpolicy(struct seg *seg,
    caddr_t addr);
static int      segkp_capable(struct seg *seg, segcapability_t capability);

/*
 * Lock used to protect the hash table(s) and caches.
 */
static kmutex_t segkp_lock;

/*
 * The segkp caches
 */
static struct segkp_cache segkp_cache[SEGKP_MAX_CACHE];

#define SEGKP_BADOP(t)  (t(*)())segkp_badop

/*
 * When there are fewer than red_minavail bytes left on the stack,
 * segkp_map_red() will map in the redzone (if called).  5000 seems
 * to work reasonably well...
 */
long            red_minavail = 5000;

/*
 * will be set to 1 for 32 bit x86 systems only, in startup.c
 */
int     segkp_fromheap = 0;
ulong_t *segkp_bitmap;

/*
```

```
 * If segkp_map_red() is called with the redzone already mapped and
 * with less than RED_DEEP_THRESHOLD bytes available on the stack,
 * then the stack situation has become quite serious;  if much more stack
 * is consumed, we have the potential of scrogging the next thread/LWP
 * structure.  To help debug the "can't happen" panics which may
 * result from this condition, we record hrestime and the calling thread
 * in red_deep_hires and red_deep_thread respectively.
 */
#define RED_DEEP_THRESHOLD      2000

hrtime_t        red_deep_hires;
kthread_t       *red_deep_thread;

uint32_t        red_nmapped;
uint32_t        red_closest = UINT_MAX;
uint32_t        red_ndoubles;

pgcnt_t anon_segkp_pages_locked;        /* See vm/anon.h */
pgcnt_t anon_segkp_pages_resv;          /* anon reserved by seg_kp */

static struct   seg_ops segkp_ops = {
        SEGKP_BADOP(int),               /* dup */
        SEGKP_BADOP(int),               /* unmap */
        SEGKP_BADOP(void),              /* free */
        segkp_fault,
        SEGKP_BADOP(faultcode_t),       /* faulta */
        SEGKP_BADOP(int),               /* setprot */
        segkp_checkprot,
        segkp_kluster,
        SEGKP_BADOP(size_t),            /* swapout */
        SEGKP_BADOP(int),               /* sync */
        SEGKP_BADOP(size_t),            /* incore */
        SEGKP_BADOP(int),               /* lockop */
        SEGKP_BADOP(int),               /* getprot */
        SEGKP_BADOP(u_offset_t),                /* getoffset */
        SEGKP_BADOP(int),               /* gettype */
        SEGKP_BADOP(int),               /* getvp */
        SEGKP_BADOP(int),               /* advise */
        segkp_dump,                     /* dump */
        segkp_pagelock,                 /* pagelock */
        SEGKP_BADOP(int),               /* setpgsz */
        segkp_getmemid,                 /* getmemid */
        segkp_getpolicy,                /* getpolicy */
        segkp_capable,                  /* capable */
        seg_inherit_notsup              /* inherit */
};
______unchanged_portion_omitted_

/*
 * segkp_map_red() will check the current frame pointer against the
 * stack base.  If the amount of stack remaining is questionable
 * (less than red_minavail), then segkp_map_red() will map in the redzone
 * and return 1.  Otherwise, it will return 0.  segkp_map_red() can
 * _only_ be called when it is safe to sleep on page_create_va().
 * _only_ be called when:
 *
 *   - it is safe to sleep on page_create_va().
 *   - the caller is non-swappable.
 *
 * It is up to the caller to remember whether segkp_map_red() successfully
 * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
 * time.
 * time.  Note that the caller must _remain_ non-swappable until after
 * calling segkp_unmap_red().
 *
 * Currently, this routine is only called from pagefault() (which necessarily
```

```
 * satisfies the above conditions).
 */
#if defined(STACK_GROWTH_DOWN)
int
segkp_map_red(void)
{
        uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
#ifndef _LP64
        caddr_t stkbase;
#endif

        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

        /*
         * Optimize for the common case where we simply return.
         */
        if ((curthread->t_red_pp == NULL) &&
            (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
                return (0);

#if defined(_LP64)
        /*
         * XXX  We probably need something better than this.
         */
        panic("kernel stack overflow");
        /*NOTREACHED*/
#else /* _LP64 */
        if (curthread->t_red_pp == NULL) {
                page_t *red_pp;
                struct seg kseg;

                caddr_t red_va = (caddr_t)
                    (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
                    PAGESIZE);

                ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
                    NULL);

                /*
                 * Allocate the physical for the red page.
                 */
                /*
                 * No PG_NORELOC here to avoid waits. Unlikely to get
                 * a relocate happening in the short time the page exists
                 * and it will be OK anyway.
                 */

                kseg.s_as = &kas;
                red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
                    PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
                ASSERT(red_pp != NULL);

                /*
                 * So we now have a page to jam into the redzone...
                 */
                page_io_unlock(red_pp);

                hat_memload(kas.a_hat, red_va, red_pp,
                    (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
                page_downgrade(red_pp);

                /*
                 * The page is left SE_SHARED locked so we can hold on to
                 * the page_t pointer.
                 */
                curthread->t_red_pp = red_pp;
                atomic_inc_32(&red_nmapped);
                while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
                        (void) atomic_cas_32(&red_closest, red_closest,
                            (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
                }
                return (1);
        }

        stkbase = (caddr_t)(((uintptr_t)curthread->t_stkbase &
            (uintptr_t)PAGEMASK) - PAGESIZE);

        atomic_inc_32(&red_ndoubles);

        if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
                /*
                 * Oh boy.  We're already deep within the mapped-in
                 * redzone page, and the caller is trying to prepare
                 * for a deep stack run.  We're running without a
                 * redzone right now:  if the caller plows off the
                 * end of the stack, it'll plow another thread or
                 * LWP structure.  That situation could result in
                 * a very hard-to-debug panic, so, in the spirit of
                 * recording the name of one's killer in one's own
                 * blood, we're going to record hrestime and the calling
                 * thread.
                 */
                red_deep_hires = hrestime.tv_nsec;
                red_deep_thread = curthread;
        }

        /*
         * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
         */
        ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
        return (0);
#endif /* _LP64 */
}

void
segkp_unmap_red(void)
{
        page_t *pp;
        caddr_t red_va = (caddr_t)(((uintptr_t)curthread->t_stkbase &
            (uintptr_t)PAGEMASK) - PAGESIZE);

        ASSERT(curthread->t_red_pp != NULL);
        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

        /*
         * Because we locked the mapping down, we can't simply rely
         * on page_destroy() to clean everything up;  we need to call
         * hat_unload() to explicitly unlock the mapping resources.
         */
        hat_unload(kas.a_hat, red_va, PAGESIZE, HAT_UNLOAD_UNLOCK);

        pp = curthread->t_red_pp;

        ASSERT(pp == page_find(&kvp, (u_offset_t)(uintptr_t)red_va));

        /*
         * Need to upgrade the SE_SHARED lock to SE_EXCL.
         */
        if (!page_tryupgrade(pp)) {
                /*
                 * As there is now wait for upgrade, release the
```

```
                 * SE_SHARED lock and wait for SE_EXCL.
                 */
                page_unlock(pp);
                pp = page_lookup(&kvp, (u_offset_t)(uintptr_t)red_va, SE_EXCL);
                /* pp may be NULL here, hence the test below */
        }

        /*
         * Destroy the page, with dontfree set to zero (i.e. free it).
         */
        if (pp != NULL)
                page_destroy(pp, 0);
        curthread->t_red_pp = NULL;
}
```
______***unchanged_portion_omitted_***

```
/*
 * Find the kpd associated with a vaddr.
 *
 * Most of the callers of segkp_find will pass the vaddr that
 * hashes to the desired index, but there are cases where
 * this is not true in which case we have to (potentially) scan
 * the whole table looking for it. This should be very rare
 * (e.g. a segkp_fault(F_INVAL) on an address somewhere in the
 * middle of the segkp_data region).
 */
static struct segkp_data *
segkp_find(struct seg *seg, caddr_t vaddr)
{
        struct segkp_segdata *kpsd = (struct segkp_segdata *)seg->s_data;
        struct segkp_data *kpd;
        int     i;
        int     stop;

        i = stop = SEGKP_HASH(vaddr);
        mutex_enter(&segkp_lock);
        do {
                for (kpd = kpsd->kpsd_hash[i]; kpd != NULL;
                    kpd = kpd->kp_next) {
                        if (vaddr >= kpd->kp_base &&
                            vaddr < kpd->kp_base + kpd->kp_len) {
                                mutex_exit(&segkp_lock);
                                return (kpd);
                        }
                }
                if (--i < 0)
                        i = SEGKP_HASHSZ - 1;   /* Wrap */
        } while (i != stop);
        mutex_exit(&segkp_lock);
        return (NULL);          /* Not found */
}

/*
 * returns size of swappable area.
 */
size_t
swapsize(caddr_t v)
{
        struct segkp_data *kpd;

        if ((kpd = segkp_find(segkp, v)) != NULL)
                return (SEGKP_MAPLEN(kpd->kp_len, kpd->kp_flags));
        else
                return (NULL);
}
```
______***unchanged_portion_omitted_***

****
    5245 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kp.h
6583 remove whole-process swapping
****

```
/*
 * 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.
 */

#ifndef _VM_SEG_KP_H
#define _VM_SEG_KP_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * segkp (as in kernel pageable) is a segment driver that supports allocation
 * of page-aligned variable size of vm resources.
 *
 * Each vm resource represents a page-aligned range of virtual addresses.
 * The caller may specify whether the resource should include a redzone,
 * be locked down, or be zero initialized.
 */

#include <vm/seg.h>
#include <sys/vmem.h>

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef  _KERNEL

/*
 * Private information per overall segkp segment (as opposed
 * to per resource within segment). There are as many anon slots
 * allocated as there there are pages in the segment.
 */
struct segkp_segdata {
        struct anon_hdr *kpsd_anon;     /* anon structs */
        vmem_t          *kpsd_arena;    /* virtual memory descriptor */
        struct segkp_data **kpsd_hash;  /* Hash table for lookups */
};
```
______***unchanged_portion_omitted_***

```
#define SEGKP_MAX_CACHE         4       /* Number of caches maintained */
```

```
/*
 * Define redzone, and stack_to_memory macros.
 * The redzone is PAGESIZE bytes.
 */
#ifdef  STACK_GROWTH_DOWN
#define KPD_REDZONE(kpd)        (0)
#define stom(v, flags)  (((flags) & KPD_HASREDZONE) ? (v) + PAGESIZE : (v))

#else   /* STACK_GROWTH_DOWN */

#define KPD_REDZONE(kpd) (btop(kpd->kp_len) - 1)
#define stom(v) (v)
#endif  /* STACK_GROWTH_DOWN */

#define SEGKP_MAPLEN(len, flags) \
        (((flags) & KPD_HASREDZONE) ? (len) - PAGESIZE : (len))

extern  struct seg *segkp;
/* If segkp becomes more than one seg this test will need changing. */
#define SEG_IS_SEGKP(SEG)       ((SEG) == segkp)

/*
 * Public routine declarations not part of the segment ops vector go here.
 */
int     segkp_create(struct seg *seg);
caddr_t segkp_get(struct seg *seg, size_t len, uint_t flags);
void    segkp_release(struct seg *seg, caddr_t vaddr);
void *  segkp_cache_init(struct seg *seg, int maxsize, size_t len,
                uint_t flags);
void    segkp_cache_free();
caddr_t segkp_cache_get(void *cookie);
int     segkp_map_red(void);
void    segkp_unmap_red(void);
size_t  swapsize(caddr_t v);

/* Special currently only used by schedctl. */
struct anon_map;        /* Make the compiler happy about the next line. */
caddr_t segkp_get_withanonmap(struct seg *, size_t, uint_t, struct anon_map *);

/*
 * We allow explicit calls to segkp_fault, even though it's part
 * of the segkp ops vector.
 */
faultcode_t segkp_fault(struct hat *hat, struct seg *seg, caddr_t addr,
        size_t len, enum fault_type type, enum seg_rw rw);

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
    9862 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kpm.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Kernel Physical Mapping (kpm) segment driver (segkpm).
 *
 * This driver delivers along with the hat_kpm* interfaces an alternative
 * mechanism for kernel mappings within the 64-bit Solaris operating system,
 * which allows the mapping of all physical memory into the kernel address
 * space at once. This is feasible in 64 bit kernels, e.g. for Ultrasparc II
 * and beyond processors, since the available VA range is much larger than
 * possible physical memory. Momentarily all physical memory is supported,
 * that is represented by the list of memory segments (memsegs).
 *
 * Segkpm mappings have also very low overhead and large pages are used
 * (when possible) to minimize the TLB and TSB footprint. It is also
 * extentable for other than Sparc architectures (e.g. AMD64). Main
 * advantage is the avoidance of the TLB-shootdown X-calls, which are
 * normally needed when a kernel (global) mapping has to be removed.
 *
 * First example of a kernel facility that uses the segkpm mapping scheme
 * is seg_map, where it is used as an alternative to hat_memload().
 * See also hat layer for more information about the hat_kpm* routines.
 * The kpm facilty can be turned off at boot time (e.g. /etc/system).
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/vnode.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/thread.h>
#include <sys/cpuvar.h>
#include <sys/bitmap.h>
#include <sys/atomic.h>
#include <sys/lgrp.h>
```

```
#include <vm/seg_kmem.h>
#include <vm/seg_kpm.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/page.h>

/*
 * Global kpm controls.
 * See also platform and mmu specific controls.
 *
 * kpm_enable -- global on/off switch for segkpm.
 * . Set by default on 64bit platforms that have kpm support.
 * . Will be disabled from platform layer if not supported.
 * . Can be disabled via /etc/system.
 *
 * kpm_smallpages -- use only regular/system pagesize for kpm mappings.
 * . Can be useful for critical debugging of kpm clients.
 * . Set to zero by default for platforms that support kpm large pages.
 *   The use of kpm large pages reduces the footprint of kpm meta data
 *   and has all the other advantages of using large pages (e.g TLB
 *   miss reduction).
 * . Set by default for platforms that don’t support kpm large pages or
 *   where large pages cannot be used for other reasons (e.g. there are
 *   only few full associative TLB entries available for large pages).
 *
 * segmap_kpm -- separate on/off switch for segmap using segkpm:
 * . Set by default.
 * . Will be disabled when kpm_enable is zero.
 * . Will be disabled when MAXBSIZE != PAGESIZE.
 * . Can be disabled via /etc/system.
 *
 */
int kpm_enable = 1;
int kpm_smallpages = 0;
int segmap_kpm = 1;

/*
 * Private seg op routines.
 */
faultcode_t segkpm_fault(struct hat *hat, struct seg *seg, caddr_t addr,
                        size_t len, enum fault_type type, enum seg_rw rw);
static void     segkpm_dump(struct seg *);
static void     segkpm_badop(void);
static int      segkpm_notsup(void);
static int      segkpm_capable(struct seg *, segcapability_t);

#define SEGKPM_BADOP(t) (t(*)())segkpm_badop
#define SEGKPM_NOTSUP   (int(*)())segkpm_notsup

static struct seg_ops segkpm_ops = {
        SEGKPM_BADOP(int),      /* dup */
        SEGKPM_BADOP(int),      /* unmap */
        SEGKPM_BADOP(void),     /* free */
        segkpm_fault,
        SEGKPM_BADOP(int),      /* faulta */
        SEGKPM_BADOP(int),      /* setprot */
        SEGKPM_BADOP(int),      /* checkprot */
        SEGKPM_BADOP(int),      /* kluster */
        SEGKPM_BADOP(size_t),   /* swapout */
        SEGKPM_BADOP(int),      /* sync */
        SEGKPM_BADOP(size_t),   /* incore */
        SEGKPM_BADOP(int),      /* lockop */
        SEGKPM_BADOP(int),      /* getprot */
        SEGKPM_BADOP(u_offset_t), /* getoffset */
        SEGKPM_BADOP(int),      /* gettype */
        SEGKPM_BADOP(int),      /* getvp */
        SEGKPM_BADOP(int),      /* advise */
        segkpm_dump,            /* dump */
        SEGKPM_NOTSUP,          /* pagelock */
        SEGKPM_BADOP(int),      /* setpgsz */
        SEGKPM_BADOP(int),      /* getmemid */
        SEGKPM_BADOP(lgrp_mem_policy_info_t *), /* getpolicy */
        segkpm_capable,         /* capable */
        seg_inherit_notsup      /* inherit */
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   58029 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/seg_map.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * VM - generic vnode mapping segment.
 *
 * The segmap driver is used only by the kernel to get faster (than seg_vn)
 * mappings [lower routine overhead; more persistent cache] to random
 * vnode/offsets.  Note than the kernel may (and does) use seg_vn as well.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/buf.h>
#include <sys/systm.h>
#include <sys/vnode.h>
#include <sys/mman.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/kmem.h>
#include <sys/vtrace.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/thread.h>
#include <sys/dumphdr.h>
#include <sys/bitmap.h>
#include <sys/lgrp.h>

#include <vm/seg_kmem.h>
```

```
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_kpm.h>
#include <vm/seg_map.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/rm.h>

/*
 * Private seg op routines.
 */
static void     segmap_free(struct seg *seg);
faultcode_t segmap_fault(struct hat *hat, struct seg *seg, caddr_t addr,
                        size_t len, enum fault_type type, enum seg_rw rw);
static faultcode_t segmap_faulta(struct seg *seg, caddr_t addr);
static int      segmap_checkprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t prot);
static int      segmap_kluster(struct seg *seg, caddr_t addr, ssize_t);
static int      segmap_getprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t *protv);
static u_offset_t       segmap_getoffset(struct seg *seg, caddr_t addr);
static int      segmap_gettype(struct seg *seg, caddr_t addr);
static int      segmap_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
static void     segmap_dump(struct seg *seg);
static int      segmap_pagelock(struct seg *seg, caddr_t addr, size_t len,
                        struct page ***ppp, enum lock_type type,
                        enum seg_rw rw);
static void     segmap_badop(void);
static int      segmap_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
static lgrp_mem_policy_info_t   *segmap_getpolicy(struct seg *seg,
    caddr_t addr);
static int      segmap_capable(struct seg *seg, segcapability_t capability);

/* segkpm support */
static caddr_t  segmap_pagecreate_kpm(struct seg *, vnode_t *, u_offset_t,
                        struct smap *, enum seg_rw);
struct smap     *get_smap_kpm(caddr_t, page_t **);

#define SEGMAP_BADOP(t) (t(*)())segmap_badop

static struct seg_ops segmap_ops = {
        SEGMAP_BADOP(int),      /* dup */
        SEGMAP_BADOP(int),      /* unmap */
        segmap_free,
        segmap_fault,
        segmap_faulta,
        SEGMAP_BADOP(int),      /* setprot */
        segmap_checkprot,
        segmap_kluster,
        SEGMAP_BADOP(size_t),   /* swapout */
        SEGMAP_BADOP(int),      /* sync */
        SEGMAP_BADOP(size_t),   /* incore */
        SEGMAP_BADOP(int),      /* lockop */
        segmap_getprot,
        segmap_getoffset,
        segmap_gettype,
        segmap_getvp,
        SEGMAP_BADOP(int),      /* advise */
        segmap_dump,
        segmap_pagelock,        /* pagelock */
        SEGMAP_BADOP(int),      /* setpgsz */
        segmap_getmemid,        /* getmemid */
        segmap_getpolicy,       /* getpolicy */
        segmap_capable,         /* capable */
        seg_inherit_notsup      /* inherit */
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   82652 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/seg_spt.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#define SEGSPT_BADOP(t) (t(*)())segspt_badop

struct seg_ops segspt_ops = {
        SEGSPT_BADOP(int),              /* dup */
        segspt_unmap,
        segspt_free,
        SEGSPT_BADOP(int),              /* fault */
        SEGSPT_BADOP(faultcode_t),      /* faulta */
        SEGSPT_BADOP(int),              /* setprot */
        SEGSPT_BADOP(int),              /* checkprot */
        SEGSPT_BADOP(int),              /* kluster */
        SEGSPT_BADOP(size_t),           /* swapout */
        SEGSPT_BADOP(int),              /* sync */
        SEGSPT_BADOP(size_t),           /* incore */
        SEGSPT_BADOP(int),              /* lockop */
        SEGSPT_BADOP(int),              /* getprot */
        SEGSPT_BADOP(u_offset_t),       /* getoffset */
        SEGSPT_BADOP(int),              /* gettype */
        SEGSPT_BADOP(int),              /* getvp */
        SEGSPT_BADOP(int),              /* advise */
        SEGSPT_BADOP(void),             /* dump */
        SEGSPT_BADOP(int),              /* pagelock */
        SEGSPT_BADOP(int),              /* setpgsz */
        SEGSPT_BADOP(int),              /* getmemid */
        segspt_getpolicy,               /* getpolicy */
        SEGSPT_BADOP(int),              /* capable */
        seg_inherit_notsup              /* inherit */
};

static int segspt_shmdup(struct seg *seg, struct seg *newseg);
static int segspt_shmunmap(struct seg *seg, caddr_t raddr, size_t ssize);
static void segspt_shmfree(struct seg *seg);
static faultcode_t segspt_shmfault(struct hat *hat, struct seg *seg,
                caddr_t addr, size_t len, enum fault_type type, enum seg_rw rw);
static faultcode_t segspt_shmfaulta(struct seg *seg, caddr_t addr);
static int segspt_shmsetprot(register struct seg *seg, register caddr_t addr,
                        register size_t len, register uint_t prot);
static int segspt_shmcheckprot(struct seg *seg, caddr_t addr, size_t size,
                        uint_t prot);
static int      segspt_shmkluster(struct seg *seg, caddr_t addr, ssize_t delta);
static size_t   segspt_shmswapout(struct seg *seg);
static size_t segspt_shmincore(struct seg *seg, caddr_t addr, size_t len,
                        register char *vec);
static int segspt_shmsync(struct seg *seg, register caddr_t addr, size_t len,
                        int attr, uint_t flags);
static int segspt_shmlockop(struct seg *seg, caddr_t addr, size_t len,
                        int attr, int op, ulong_t *lockmap, size_t pos);
static int segspt_shmgetprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t *protv);
static u_offset_t segspt_shmgetoffset(struct seg *seg, caddr_t addr);
static int segspt_shmgettype(struct seg *seg, caddr_t addr);
static int segspt_shmgetvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
static int segspt_shmadvise(struct seg *seg, caddr_t addr, size_t len,
                        uint_t behav);
static void segspt_shmdump(struct seg *seg);
static int segspt_shmpagelock(struct seg *, caddr_t, size_t,
                        struct page ***, enum lock_type, enum seg_rw);
static int segspt_shmsetpgsz(struct seg *, caddr_t, size_t, uint_t);
static int segspt_shmgetmemid(struct seg *, caddr_t, memid_t *);
```

```
static lgrp_mem_policy_info_t *segspt_shmgetpolicy(struct seg *, caddr_t);
static int segspt_shmcapable(struct seg *, segcapability_t);

struct seg_ops segspt_shmops = {
        segspt_shmdup,
        segspt_shmunmap,
        segspt_shmfree,
        segspt_shmfault,
        segspt_shmfaulta,
        segspt_shmsetprot,
        segspt_shmcheckprot,
        segspt_shmkluster,
        segspt_shmswapout,
        segspt_shmsync,
        segspt_shmincore,
        segspt_shmlockop,
        segspt_shmgetprot,
        segspt_shmgetoffset,
        segspt_shmgettype,
        segspt_shmgetvp,
        segspt_shmadvise,       /* advise */
        segspt_shmdump,
        segspt_shmpagelock,
        segspt_shmsetpgsz,
        segspt_shmgetmemid,
        segspt_shmgetpolicy,
        segspt_shmcapable,
        seg_inherit_notsup
};
______unchanged_portion_omitted_

/*ARGSUSED*/
static int
segspt_shmkluster(struct seg *seg, caddr_t addr, ssize_t delta)
{
        return (0);
}

/*ARGSUSED*/
static size_t
segspt_shmswapout(struct seg *seg)
{
        return (0);
}
______unchanged_portion_omitted_
```

****
 279551 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/seg_vn.c
6583 remove whole-process swapping
****

```
/*
 * 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 (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - shared or copy-on-write from a vnode/anonymous memory.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/debug.h>
#include <sys/cred.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/bitmap.h>
#include <sys/swap.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/cmn_err.h>
#include <sys/callb.h>
#include <sys/vm.h>
```

```
#include <sys/dumphdr.h>
#include <sys/lgrp.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_vn.h>
#include <vm/pvn.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/vpage.h>
#include <sys/proc.h>
#include <sys/task.h>
#include <sys/project.h>
#include <sys/zone.h>
#include <sys/shm_impl.h>

/*
 * segvn_fault needs a temporary page list array.  To avoid calling kmem all
 * the time, it creates a small (PVN_GETPAGE_NUM entry) array and uses it if
 * it can.  In the rare case when this page list is not large enough, it
 * goes and gets a large enough array from kmem.
 *
 * This small page list array covers either 8 pages or 64kB worth of pages -
 * whichever is smaller.
 */
#define PVN_MAX_GETPAGE_SZ      0x10000
#define PVN_MAX_GETPAGE_NUM     0x8

#if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
#define PVN_GETPAGE_SZ  ptob(PVN_MAX_GETPAGE_NUM)
#define PVN_GETPAGE_NUM PVN_MAX_GETPAGE_NUM
#else
#define PVN_GETPAGE_SZ  PVN_MAX_GETPAGE_SZ
#define PVN_GETPAGE_NUM btop(PVN_MAX_GETPAGE_SZ)
#endif

/*
 * Private seg op routines.
 */
static int      segvn_dup(struct seg *seg, struct seg *newseg);
static int      segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
static void     segvn_free(struct seg *seg);
static faultcode_t segvn_fault(struct hat *hat, struct seg *seg,
                    caddr_t addr, size_t len, enum fault_type type,
                    enum seg_rw rw);
static faultcode_t segvn_faulta(struct seg *seg, caddr_t addr);
static int      segvn_setprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static int      segvn_checkprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
static size_t   segvn_swapout(struct seg *seg);
static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
                    int attr, uint_t flags);
static size_t   segvn_incore(struct seg *seg, caddr_t addr, size_t len,
                    char *vec);
static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
                    int attr, int op, ulong_t *lockmap, size_t pos);
static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
                    uint_t *protv);
static u_offset_t       segvn_getoffset(struct seg *seg, caddr_t addr);
static int      segvn_gettype(struct seg *seg, caddr_t addr);
static int      segvn_getvp(struct seg *seg, caddr_t addr,
                    struct vnode **vpp);
static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,
                    uint_t behav);
static void     segvn_dump(struct seg *seg);
static int      segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
                    struct page ***ppp, enum lock_type type, enum seg_rw rw);
static int      segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
                    uint_t szc);
static int      segvn_getmemid(struct seg *seg, caddr_t addr,
                    memid_t *memidp);
static lgrp_mem_policy_info_t   *segvn_getpolicy(struct seg *, caddr_t);
static int      segvn_capable(struct seg *seg, segcapability_t capable);
static int      segvn_inherit(struct seg *, caddr_t, size_t, uint_t);

struct  seg_ops segvn_ops = {
        segvn_dup,
        segvn_unmap,
        segvn_free,
        segvn_fault,
        segvn_faulta,
        segvn_setprot,
        segvn_checkprot,
        segvn_kluster,
        segvn_swapout,
        segvn_sync,
        segvn_incore,
        segvn_lockop,
        segvn_getprot,
        segvn_getoffset,
        segvn_gettype,
        segvn_getvp,
        segvn_advise,
        segvn_dump,
        segvn_pagelock,
        segvn_setpagesize,
        segvn_getmemid,
        segvn_getpolicy,
        segvn_capable,
        segvn_inherit
};
______unchanged_portion_omitted_

/*
 * Check to see if it makes sense to do kluster/read ahead to
 * addr + delta relative to the mapping at addr.  We assume here
 * that delta is a signed PAGESIZE'd multiple (which can be negative).
 *
 * For segvn, we currently "approve" of the action if we are
 * still in the segment and it maps from the same vp/off,
 * or if the advice stored in segvn_data or vpages allows it.
 * Currently, klustering is not allowed only if MADV_RANDOM is set.
 */
static int
segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon *oap, *ap;
        ssize_t pd;
        size_t page;
        struct vnode *vp1, *vp2;
        u_offset_t off1, off2;
        struct anon_map *amp;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
        ASSERT(AS_WRITE_HELD(seg->s_as) ||
            SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

        if (addr + delta < seg->s_base ||
```

```
            addr + delta >= (seg->s_base + seg->s_size))
                return (-1);            /* exceeded segment bounds */

        pd = delta / (ssize_t)PAGESIZE; /* divide to preserve sign bit */
        page = seg_page(seg, addr);

        /*
         * Check to see if either of the pages addr or addr + delta
         * have advice set that prevents klustering (if MADV_RANDOM advice
         * is set for entire segment, or MADV_SEQUENTIAL is set and delta
         * is negative).
         */
        if (svd->advice == MADV_RANDOM ||
            svd->advice == MADV_SEQUENTIAL && delta < 0)
                return (-1);
        else if (svd->pageadvice && svd->vpage) {
                struct vpage *bvpp, *evpp;

                bvpp = &svd->vpage[page];
                evpp = &svd->vpage[page + pd];
                if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
                    VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
                        return (-1);
                if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
                    VPP_ADVICE(evpp) == MADV_RANDOM)
                        return (-1);
        }

        if (svd->type == MAP_SHARED)
                return (0);             /* shared mapping - all ok */

        if ((amp = svd->amp) == NULL)
                return (0);             /* off original vnode */

        page += svd->anon_index;

        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

        oap = anon_get_ptr(amp->ahp, page);
        ap = anon_get_ptr(amp->ahp, page + pd);

        ANON_LOCK_EXIT(&amp->a_rwlock);

        if ((oap == NULL && ap != NULL) || (oap != NULL && ap == NULL)) {
                return (-1);            /* one with and one without an anon */
        }

        if (oap == NULL) {              /* implies that ap == NULL */
                return (0);             /* off original vnode */
        }

        /*
         * Now we know we have two anon pointers - check to
         * see if they happen to be properly allocated.
         */

        /*
         * XXX We cheat here and don't lock the anon slots. We can't because
         * we may have been called from the anon layer which might already
         * have locked them. We are holding a refcnt on the slots so they
         * can't disappear. The worst that will happen is we'll get the wrong
         * names (vp, off) for the slots and make a poor klustering decision.
         */
        swap_xlate(ap, &vp1, &off1);
        swap_xlate(oap, &vp2, &off2);
        if (!VOP_CMP(vp1, vp2, NULL) || off1 - off2 != delta)
                return (-1);
        return (0);
}

/*
 * Swap the pages of seg out to secondary storage, returning the
 * number of bytes of storage freed.
 *
 * The basic idea is first to unload all translations and then to call
 * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
 * swap device.  Pages to which other segments have mappings will remain
 * mapped and won't be swapped.  Our caller (as_swapout) has already
 * performed the unloading step.
 *
 * The value returned is intended to correlate well with the process's
 * memory requirements.  However, there are some caveats:
 * 1)   When given a shared segment as argument, this routine will
 *      only succeed in swapping out pages for the last sharer of the
 *      segment.  (Previous callers will only have decremented mapping
 *      reference counts.)
 * 2)   We assume that the hat layer maintains a large enough translation
 *      cache to capture process reference patterns.
 */
static size_t
segvn_swapout(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp;
        pgcnt_t pgcnt = 0;
        pgcnt_t npages;
        pgcnt_t page;
        ulong_t anon_index;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        /*
         * Find pages unmapped by our caller and force them
         * out to the virtual swap device.
         */
        if ((amp = svd->amp) != NULL)
                anon_index = svd->anon_index;
        npages = seg->s_size >> PAGESHIFT;
        for (page = 0; page < npages; page++) {
                page_t *pp;
                struct anon *ap;
                struct vnode *vp;
                u_offset_t off;
                anon_sync_obj_t cookie;

                /*
                 * Obtain <vp, off> pair for the page, then look it up.
                 *
                 * Note that this code is willing to consider regular
                 * pages as well as anon pages.  Is this appropriate here?
                 */
                ap = NULL;
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        if (anon_array_try_enter(amp, anon_index + page,
                            &cookie)) {
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                continue;
                        }
```

```
                                ap = anon_get_ptr(amp->ahp, anon_index + page);
                                if (ap != NULL) {
                                        swap_xlate(ap, &vp, &off);
                                } else {
                                        vp = svd->vp;
                                        off = svd->offset + ptob(page);
                                }
                                anon_array_exit(&cookie);
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
                        vp = svd->vp;
                        off = svd->offset + ptob(page);
                }
                if (vp == NULL) {               /* untouched zfod page */
                        ASSERT(ap == NULL);
                        continue;
                }

                pp = page_lookup_nowait(vp, off, SE_SHARED);
                if (pp == NULL)
                        continue;


                /*
                 * Examine the page to see whether it can be tossed out,
                 * keeping track of how many we've found.
                 */
                if (!page_tryupgrade(pp)) {
                        /*
                         * If the page has an i/o lock and no mappings,
                         * it's very likely that the page is being
                         * written out as a result of klustering.
                         * Assume this is so and take credit for it here.
                         */
                        if (!page_io_trylock(pp)) {
                                if (!hat_page_is_mapped(pp))
                                        pgcnt++;
                        } else {
                                page_io_unlock(pp);
                        }
                        page_unlock(pp);
                        continue;
                }
                ASSERT(!page_iolock_assert(pp));


                /*
                 * Skip if page is locked or has mappings.
                 * We don't need the page_struct_lock to look at lckcnt
                 * and cowcnt because the page is exclusive locked.
                 */
                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0 ||
                    hat_page_is_mapped(pp)) {
                        page_unlock(pp);
                        continue;
                }

                /*
                 * dispose skips large pages so try to demote first.
                 */
                if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
                        page_unlock(pp);
                        /*
                         * XXX should skip the remaining page_t's of this
                         * large page.
                         */
```

```
				continue;
			}

			ASSERT(pp->p_szc == 0);

			/*
			 * No longer mapped -- we can toss it out.  How
			 * we do so depends on whether or not it's dirty.
			 */
			if (hat_ismod(pp) && pp->p_vnode) {
				/*
				 * We must clean the page before it can be
				 * freed.  Setting B_FREE will cause pvn_done
				 * to free the page when the i/o completes.
				 * XXX: This also causes it to be accounted
				 *      as a pageout instead of a swap: need
				 *      B_SWAPOUT bit to use instead of B_FREE.
				 *
				 * Hold the vnode before releasing the page lock
				 * to prevent it from being freed and re-used by
				 * some other thread.
				 */
				VN_HOLD(vp);
				page_unlock(pp);

				/*
				 * Queue all i/o requests for the pageout thread
				 * to avoid saturating the pageout devices.
				 */
				if (!queue_io_request(vp, off))
					VN_RELE(vp);
			} else {
				/*
				 * The page was clean, free it.
				 *
				 * XXX: Can we ever encounter modified pages
				 *      with no associated vnode here?
				 */
				ASSERT(pp->p_vnode != NULL);
				/*LINTED: constant in conditional context*/
				VN_DISPOSE(pp, B_FREE, 0, kcred);
			}

			/*
			 * Credit now even if i/o is in progress.
			 */
			pgcnt++;
		}
		SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

		/*
		 * Wakeup pageout to initiate i/o on all queued requests.
		 */
		cv_signal_pageout();
		return (ptob(pgcnt));
}
```
______**_unchanged_portion_omitted_**

****
   89741 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/vm_anon.c
6583 remove whole-process swapping
****

```
/*
 * 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 (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - anonymous pages.
 *
 * This layer sits immediately above the vm_swap layer.  It manages
 * physical pages that have no permanent identity in the file system
 * name space, using the services of the vm_swap layer to allocate
 * backing storage for these pages.  Since these pages have no external
 * identity, they are discarded when the last reference is removed.
 *
 * An important function of this layer is to manage low-level sharing
 * of pages that are logically distinct but that happen to be
 * physically identical (e.g., the corresponding pages of the processes
 * resulting from a fork before one process or the other changes their
 * contents).  This pseudo-sharing is present only as an optimization
 * and is not to be confused with true sharing in which multiple
 * address spaces deliberately contain references to the same object;
 * such sharing is managed at a higher level.
 *
 * The key data structure here is the anon struct, which contains a
 * reference count for its associated physical page and a hint about
 * the identity of that page.  Anon structs typically live in arrays,
 * with an instance’s position in its array determining where the
 * corresponding backing storage is allocated; however, the swap_xlate()
 * routine abstracts away this representation information so that the
```

```
 * rest of the anon layer need not know it.  (See the swap layer for
 * more details on anon struct layout.)
 *
 * In the future versions of the system, the association between an
 * anon struct and its position on backing store will change so that
 * we don’t require backing store all anonymous pages in the system.
 * This is important for consideration for large memory systems.
 * We can also use this technique to delay binding physical locations
 * to anonymous pages until pageout time where we can make smarter
 * allocation decisions to improve anonymous klustering.
 * to anonymous pages until pageout/swapout time where we can make
 * smarter allocation decisions to improve anonymous klustering.
 *
 * Many of the routines defined here take a (struct anon **) argument,
 * which allows the code at this level to manage anon pages directly,
 * so that callers can regard anon structs as opaque objects and not be
 * concerned with assigning or inspecting their contents.
 *
 * Clients of this layer refer to anon pages indirectly.  That is, they
 * maintain arrays of pointers to anon structs rather than maintaining
 * anon structs themselves.  The (struct anon **) arguments mentioned
 * above are pointers to entries in these arrays.  It is these arrays
 * that capture the mapping between offsets within a given segment and
 * the corresponding anonymous backing storage address.
 */

#ifdef DEBUG
#define ANON_DEBUG
#endif

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/cred.h>
#include <sys/thread.h>
#include <sys/vnode.h>
#include <sys/cpuvar.h>
#include <sys/swap.h>
#include <sys/cmn_err.h>
#include <sys/vtrace.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/bitmap.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/debug.h>
#include <sys/fs/swapnode.h>
#include <sys/tnf_probe.h>
#include <sys/lgrp.h>
#include <sys/policy.h>
#include <sys/condvar_impl.h>
#include <sys/mutex_impl.h>
#include <sys/rctl.h>

#include <vm/as.h>
#include <vm/hat.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/vpage.h>
#include <vm/seg.h>
#include <vm/rm.h>

#include <fs/fs_subr.h>
```

```
struct vnode *anon_vp;

int anon_debug;

kmutex_t        anoninfo_lock;
struct          k_anoninfo k_anoninfo;
ani_free_t      *ani_free_pool;
pad_mutex_t     anon_array_lock[ANON_LOCKSIZE];
kcondvar_t      anon_array_cv[ANON_LOCKSIZE];

/*
 * Global hash table for (vp, off) -> anon slot
 */
extern  int swap_maxcontig;
size_t  anon_hash_size;
unsigned int anon_hash_shift;
struct anon **anon_hash;

static struct kmem_cache *anon_cache;
static struct kmem_cache *anonmap_cache;

pad_mutex_t     *anonhash_lock;

/*
 * Used to make the increment of all refcnts of all anon slots of a large
 * page appear to be atomic.  The lock is grabbed for the first anon slot of
 * a large page.
 */
pad_mutex_t     *anonpages_hash_lock;

#define APH_MUTEX(vp, off)                              \
        (&anonpages_hash_lock[(ANON_HASH((vp), (off)) & \
            (AH_LOCK_SIZE - 1))].pad_mutex)

#ifdef VM_STATS
static struct anonvmstats_str {
        ulong_t getpages[30];
        ulong_t privatepages[10];
        ulong_t demotepages[9];
        ulong_t decrefpages[9];
        ulong_t dupfillholes[4];
        ulong_t freepages[1];
} anonvmstats;
______unchanged_portion_omitted_

void
anon_array_enter(struct anon_map *amp, ulong_t an_idx, anon_sync_obj_t *sobj)
{
        ulong_t         *ap_slot;
        kmutex_t        *mtx;
        kcondvar_t      *cv;
        int             hash;

        /*
         * Use szc to determine anon slot(s) to appear atomic.
         * If szc = 0, then lock the anon slot and mark it busy.
         * If szc > 0, then lock the range of slots by getting the
         * anon_array_lock for the first anon slot, and mark only the
         * first anon slot busy to represent whole range being busy.
         */

        ASSERT(RW_READ_HELD(&amp->a_rwlock));
        an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
        hash = ANON_ARRAY_HASH(amp, an_idx);
        sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
        sobj->sync_cv = cv = &anon_array_cv[hash];
```

```
        mutex_enter(mtx);
        ap_slot = anon_get_slot(amp->ahp, an_idx);
        while (ANON_ISBUSY(ap_slot))
                cv_wait(cv, mtx);
        ANON_SETBUSY(ap_slot);
        sobj->sync_data = ap_slot;
        mutex_exit(mtx);
}

int
anon_array_try_enter(struct anon_map *amp, ulong_t an_idx,
                        anon_sync_obj_t *sobj)
{
        ulong_t         *ap_slot;
        kmutex_t        *mtx;
        int             hash;

        /*
         * Try to lock a range of anon slots.
         * Use szc to determine anon slot(s) to appear atomic.
         * If szc = 0, then lock the anon slot and mark it busy.
         * If szc > 0, then lock the range of slots by getting the
         * anon_array_lock for the first anon slot, and mark only the
         * first anon slot busy to represent whole range being busy.
         * Fail if the mutex or the anon_array are busy.
         */

        ASSERT(RW_READ_HELD(&amp->a_rwlock));
        an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
        hash = ANON_ARRAY_HASH(amp, an_idx);
        sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
        sobj->sync_cv = &anon_array_cv[hash];
        if (!mutex_tryenter(mtx)) {
                return (EWOULDBLOCK);
        }
        ap_slot = anon_get_slot(amp->ahp, an_idx);
        if (ANON_ISBUSY(ap_slot)) {
                mutex_exit(mtx);
                return (EWOULDBLOCK);
        }
        ANON_SETBUSY(ap_slot);
        sobj->sync_data = ap_slot;
        mutex_exit(mtx);
        return (0);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   88974 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/vm_as.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*
 * Swap the pages associated with the address space as out to
 * secondary storage, returning the number of bytes actually
 * swapped.
 *
 * The value returned is intended to correlate well with the process's
 * memory requirements.  Its usefulness for this purpose depends on
 * how well the segment-level routines do at returning accurate
 * information.
 */
size_t
as_swapout(struct as *as)
{
        struct seg *seg;
        size_t swpcnt = 0;

        /*
         * Kernel-only processes have given up their address
         * spaces.  Of course, we shouldn't be attempting to
         * swap out such processes in the first place...
         */
        if (as == NULL)
                return (0);

        AS_LOCK_ENTER(as, RW_READER);

        /*
         * Free all mapping resources associated with the address
         * space.  The segment-level swapout routines capitalize
         * on this unmapping by scavanging pages that have become
         * unmapped here.
         */
        hat_swapout(as->a_hat);

        /*
         * Call the swapout routines of all segments in the address
         * space to do the actual work, accumulating the amount of
         * space reclaimed.
         */
        for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
                struct seg_ops *ov = seg->s_ops;

                /*
                 * We have to check to see if the seg has
                 * an ops vector because the seg may have
                 * been in the middle of being set up when
                 * the process was picked for swapout.
                 */
                if ((ov != NULL) && (ov->swapout != NULL))
                        swpcnt += SEGOP_SWAPOUT(seg);
        }
        AS_LOCK_EXIT(as);
        return (swpcnt);
}

/*
 * Determine whether data from the mappings in interval [addr, addr + size)
 * are in the primary memory (core) cache.
 */
```

```
int
as_incore(struct as *as, caddr_t addr,
    size_t size, char *vec, size_t *sizep)
{
        struct seg *seg;
        size_t ssize;
        caddr_t raddr;          /* rounded down addr */
        size_t rsize;           /* rounded up size */
        size_t isize;                   /* iteration size */
        int error = 0;          /* result, assume success */

        *sizep = 0;
        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
        rsize = ((((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
            (size_t)raddr;

        if (raddr + rsize < raddr)              /* check for wraparound */
                return (ENOMEM);

        AS_LOCK_ENTER(as, RW_READER);
        seg = as_segat(as, raddr);
        if (seg == NULL) {
                AS_LOCK_EXIT(as);
                return (-1);
        }

        for (; rsize != 0; rsize -= ssize, raddr += ssize) {
                if (raddr >= seg->s_base + seg->s_size) {
                        seg = AS_SEGNEXT(as, seg);
                        if (seg == NULL || raddr != seg->s_base) {
                                error = -1;
                                break;
                        }
                }
                if ((raddr + rsize) > (seg->s_base + seg->s_size))
                        ssize = seg->s_base + seg->s_size - raddr;
                else
                        ssize = rsize;
                *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
                if (isize != ssize) {
                        error = -1;
                        break;
                }
                vec += btopr(ssize);
        }
        AS_LOCK_EXIT(as);
        return (error);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   13931 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/os/mlsetup.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*
 * Setup routine called right before main(). Interposing this function
 * before main() allows us to call it in a machine-independent fashion.
 */
void
mlsetup(struct regs *rp)
{
        u_longlong_t prop_value;
        extern struct classfuncs sys_classfuncs;
        extern disp_t cpu0_disp;
        extern char t0stack[];
        extern int post_fastreboot;
        extern uint64_t plat_dr_options;

        ASSERT_STACK_ALIGNED();

        /*
         * initialize cpu_self
         */
        cpu[0]->cpu_self = cpu[0];

#if defined(__xpv)
        /*
         * Point at the hypervisor's virtual cpu structure
         */
        cpu[0]->cpu_m.mcpu_vcpu_info = &HYPERVISOR_shared_info->vcpu_info[0];
#endif

        /*
         * check if we've got special bits to clear or set
         * when checking cpu features
         */

        if (bootprop_getval("cpuid_feature_ecx_include", &prop_value) != 0)
                cpuid_feature_ecx_include = 0;
        else
                cpuid_feature_ecx_include = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_ecx_exclude", &prop_value) != 0)
                cpuid_feature_ecx_exclude = 0;
        else
                cpuid_feature_ecx_exclude = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_edx_include", &prop_value) != 0)
                cpuid_feature_edx_include = 0;
        else
                cpuid_feature_edx_include = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_edx_exclude", &prop_value) != 0)
                cpuid_feature_edx_exclude = 0;
        else
                cpuid_feature_edx_exclude = (uint32_t)prop_value;

        /*
         * Initialize idt0, gdt0, ldt0_default, ktss0 and dftss.
         */
        init_desctbls();
```

```
        /*
         * lgrp_init() and possibly cpuid_pass1() need PCI config
         * space access
         */
#if defined(__xpv)
        if (DOMAIN_IS_INITDOMAIN(xen_info))
                pci_cfgspace_init();
#else
        pci_cfgspace_init();
        /*
         * Initialize the platform type from CPU 0 to ensure that
         * determine_platform() is only ever called once.
         */
        determine_platform();
#endif

        /*
         * The first lightweight pass (pass0) through the cpuid data
         * was done in locore before mlsetup was called.  Do the next
         * pass in C code.
         *
         * The x86_featureset is initialized here based on the capabilities
         * of the boot CPU.  Note that if we choose to support CPUs that have
         * different feature sets (at which point we would almost certainly
         * want to set the feature bits to correspond to the feature
         * minimum) this value may be altered.
         */
        cpuid_pass1(cpu[0], x86_featureset);

#if !defined(__xpv)
        if ((get_hwenv() & HW_XEN_HVM) != 0)
                xen_hvm_init();

        /*
         * Before we do anything with the TSCs, we need to work around
         * Intel erratum BT81.  On some CPUs, warm reset does not
         * clear the TSC.  If we are on such a CPU, we will clear TSC ourselves
         * here.  Other CPUs will clear it when we boot them later, and the
         * resulting skew will be handled by tsc_sync_master()/_slave();
         * note that such skew already exists and has to be handled anyway.
         *
         * We do this only on metal.  This same problem can occur with a
         * hypervisor that does not happen to virtualise a TSC that starts from
         * zero, regardless of CPU type; however, we do not expect hypervisors
         * that do not virtualise TSC that way to handle writes to TSC
         * correctly, either.
         */
        if (get_hwenv() == HW_NATIVE &&
            cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
            cpuid_getfamily(CPU) == 6 &&
            (cpuid_getmodel(CPU) == 0x2d || cpuid_getmodel(CPU) == 0x3e) &&
            is_x86_feature(x86_featureset, X86FSET_TSC)) {
                (void) wrmsr(REG_TSC, 0UL);
        }

        /*
         * Patch the tsc_read routine with appropriate set of instructions,
         * depending on the processor family and architecure, to read the
         * time-stamp counter while ensuring no out-of-order execution.
         * Patch it while the kernel text is still writable.
         *
         * Note: tsc_read is not patched for intel processors whose family
         * is >6 and for amd whose family >f (in case they don't support rdtscp
         * instruction, unlikely). By default tsc_read will use cpuid for
         * serialization in such cases. The following code needs to be
         * revisited if intel processors of family >= f retains the
         * instruction serialization nature of mfence instruction.
         * Note: tsc_read is not patched for x86 processors which do
         * not support "mfence". By default tsc_read will use cpuid for
         * serialization in such cases.
         *
         * The Xen hypervisor does not correctly report whether rdtscp is
         * supported or not, so we must assume that it is not.
         */
        if ((get_hwenv() & HW_XEN_HVM) == 0 &&
            is_x86_feature(x86_featureset, X86FSET_TSCP))
                patch_tsc_read(X86_HAVE_TSCP);
        else if (cpuid_getvendor(CPU) == X86_VENDOR_AMD &&
            cpuid_getfamily(CPU) <= 0xf &&
            is_x86_feature(x86_featureset, X86FSET_SSE2))
                patch_tsc_read(X86_TSC_MFENCE);
        else if (cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
            cpuid_getfamily(CPU) <= 6 &&
            is_x86_feature(x86_featureset, X86FSET_SSE2))
                patch_tsc_read(X86_TSC_LFENCE);

#endif  /* !__xpv */

#if defined(__i386) && !defined(__xpv)
        /*
         * Some i386 processors do not implement the rdtsc instruction,
         * or at least they do not implement it correctly. Patch them to
         * return 0.
         */
        if (!is_x86_feature(x86_featureset, X86FSET_TSC))
                patch_tsc_read(X86_NO_TSC);
#endif  /* __i386 && !__xpv */

#if defined(__amd64) && !defined(__xpv)
        patch_memops(cpuid_getvendor(CPU));
#endif  /* __amd64 && !__xpv */

#if !defined(__xpv)
        /* XXPV what, if anything, should be dorked with here under xen? */

        /*
         * While we're thinking about the TSC, let's set up %cr4 so that
         * userland can issue rdtsc, and initialize the TSC_AUX value
         * (the cpuid) for the rdtscp instruction on appropriately
         * capable hardware.
         */
        if (is_x86_feature(x86_featureset, X86FSET_TSC))
                setcr4(getcr4() & ~CR4_TSD);

        if (is_x86_feature(x86_featureset, X86FSET_TSCP))
                (void) wrmsr(MSR_AMD_TSCAUX, 0);

        /*
         * Let's get the other %cr4 stuff while we're here.
         */
        if (is_x86_feature(x86_featureset, X86FSET_DE))
                setcr4(getcr4() | CR4_DE);

        if (is_x86_feature(x86_featureset, X86FSET_SMEP))
                setcr4(getcr4() | CR4_SMEP);
#endif /* __xpv */

        /*
         * initialize t0
         */
        t0.t_stk = (caddr_t)rp - MINFRAME;
        t0.t_stkbase = t0stack;
```

```
        t0.t_pri = maxclsyspri - 3;
        t0.t_schedflag = 0;
        t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
        t0.t_procp = &p0;
        t0.t_plockp = &p0lock.pl_lock;
        t0.t_lwp = &lwp0;
        t0.t_forw = &t0;
        t0.t_back = &t0;
        t0.t_next = &t0;
        t0.t_prev = &t0;
        t0.t_cpu = cpu[0];
        t0.t_disp_queue = &cpu0_disp;
        t0.t_bind_cpu = PBIND_NONE;
        t0.t_bind_pset = PS_NONE;
        t0.t_bindflag = (uchar_t)default_binding_mode;
        t0.t_cpupart = &cp_default;
        t0.t_clfuncs = &sys_classfuncs.thread;
        t0.t_copyops = NULL;
        THREAD_ONPROC(&t0, CPU);

        lwp0.lwp_thread = &t0;
        lwp0.lwp_regs = (void *)rp;
        lwp0.lwp_procp = &p0;
        t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

        p0.p_exec = NULL;
        p0.p_stat = SRUN;
        p0.p_flag = SSYS;
        p0.p_tlist = &t0;
        p0.p_stksize = 2*PAGESIZE;
        p0.p_stkpageszc = 0;
        p0.p_as = &kas;
        p0.p_lockp = &p0lock;
        p0.p_brkpageszc = 0;
        p0.p_t1_lgrpid = LGRP_NONE;
        p0.p_tr_lgrpid = LGRP_NONE;
        sigorset(&p0.p_ignore, &ignoredefault);

        CPU->cpu_thread = &t0;
        bzero(&cpu0_disp, sizeof (disp_t));
        CPU->cpu_disp = &cpu0_disp;
        CPU->cpu_disp->disp_cpu = CPU;
        CPU->cpu_dispthread = &t0;
        CPU->cpu_idle_thread = &t0;
        CPU->cpu_flags = CPU_READY | CPU_RUNNING | CPU_EXISTS | CPU_ENABLE;
        CPU->cpu_dispatch_pri = t0.t_pri;

        CPU->cpu_id = 0;

        CPU->cpu_pri = 12;              /* initial PIL for the boot CPU */

        /*
         * The kernel doesn't use LDTs unless a process explicitly requests one.
         */
        p0.p_ldt_desc = null_sdesc;

        /*
         * Initialize thread/cpu microstate accounting
         */
        init_mstate(&t0, LMS_SYSTEM);
        init_cpu_mstate(CPU, CMS_SYSTEM);

        /*
         * Initialize lists of available and active CPUs.
         */
        cpu_list_init(CPU);
        pg_cpu_bootstrap(CPU);

        /*
         * Now that we have taken over the GDT, IDT and have initialized
         * active CPU list it's time to inform kmdb if present.
         */
        if (boothowto & RB_DEBUG)
                kdi_idt_sync();

        /*
         * Explicitly set console to text mode (0x3) if this is a boot
         * post Fast Reboot, and the console is set to CONS_SCREEN_TEXT.
         */
        if (post_fastreboot && boot_console_type(NULL) == CONS_SCREEN_TEXT)
                set_console_mode(0x3);

        /*
         * If requested (boot -d) drop into kmdb.
         *
         * This must be done after cpu_list_init() on the 64-bit kernel
         * since taking a trap requires that we re-compute gsbase based
         * on the cpu list.
         */
        if (boothowto & RB_DEBUGENTER)
                kmdb_enter();

        cpu_vm_data_init(CPU);

        rp->r_fp = 0;   /* terminate kernel stack traces! */

        prom_init("kernel", (void *)NULL);

        /* User-set option overrides firmware value. */
        if (bootprop_getval(PLAT_DR_OPTIONS_NAME, &prop_value) == 0) {
                plat_dr_options = (uint64_t)prop_value;
        }
#if defined(__xpv)
        /* No support of DR operations on xpv */
        plat_dr_options = 0;
#else   /* __xpv */
        /* Flag PLAT_DR_FEATURE_ENABLED should only be set by DR driver. */
        plat_dr_options &= ~PLAT_DR_FEATURE_ENABLED;
#ifndef __amd64
        /* Only enable CPU/memory DR on 64 bits kernel. */
        plat_dr_options &= ~PLAT_DR_FEATURE_MEMORY;
        plat_dr_options &= ~PLAT_DR_FEATURE_CPU;
#endif  /* __amd64 */
#endif  /* __xpv */

        /*
         * Get value of "plat_dr_physmax" boot option.
         * It overrides values calculated from MSCT or SRAT table.
         */
        if (bootprop_getval(PLAT_DR_PHYSMAX_NAME, &prop_value) == 0) {
                plat_dr_physmax = ((uint64_t)prop_value) >> PAGESHIFT;
        }

        /* Get value of boot_ncpus. */
        if (bootprop_getval(BOOT_NCPUS_NAME, &prop_value) != 0) {
                boot_ncpus = NCPU;
        } else {
                boot_ncpus = (int)prop_value;
                if (boot_ncpus <= 0 || boot_ncpus > NCPU)
                        boot_ncpus = NCPU;
        }
```

```
        /*
         * Set max_ncpus and boot_max_ncpus to boot_ncpus if platform doesn't
         * support CPU DR operations.
         */
        if (plat_dr_support_cpu() == 0) {
                max_ncpus = boot_max_ncpus = boot_ncpus;
        } else {
                if (bootprop_getval(PLAT_MAX_NCPUS_NAME, &prop_value) != 0) {
                        max_ncpus = NCPU;
                } else {
                        max_ncpus = (int)prop_value;
                        if (max_ncpus <= 0 || max_ncpus > NCPU) {
                                max_ncpus = NCPU;
                        }
                        if (boot_ncpus > max_ncpus) {
                                boot_ncpus = max_ncpus;
                        }
                }

                if (bootprop_getval(BOOT_MAX_NCPUS_NAME, &prop_value) != 0) {
                        boot_max_ncpus = boot_ncpus;
                } else {
                        boot_max_ncpus = (int)prop_value;
                        if (boot_max_ncpus <= 0 || boot_max_ncpus > NCPU) {
                                boot_max_ncpus = boot_ncpus;
                        } else if (boot_max_ncpus > max_ncpus) {
                                boot_max_ncpus = max_ncpus;
                        }
                }
        }

        /*
         * Initialize the lgrp framework
         */
        lgrp_init(LGRP_INIT_STAGE1);

        if (boothowto & RB_HALT) {
                prom_printf("unix: kernel halted by -h flag\n");
                prom_enter_mon();
        }

        ASSERT_STACK_ALIGNED();

        /*
         * Fill out cpu_ucode_info.  Update microcode if necessary.
         */
        ucode_check(CPU);

        if (workaround_errata(CPU) != 0)
                panic("critical workaround(s) missing for boot cpu");
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   61922 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/os/trap.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#endif /* OPTERON_ERRATUM_91 */

/*
 * Called from the trap handler when a processor trap occurs.
 *
 * Note: All user-level traps that might call stop() must exit
 * trap() by 'goto out' or by falling through.
 * Note Also: trap() is usually called with interrupts enabled, (PS_IE == 1)
 * however, there are paths that arrive here with PS_IE == 0 so special care
 * must be taken in those cases.
 */
void
trap(struct regs *rp, caddr_t addr, processorid_t cpuid)
{
        kthread_t *ct = curthread;
        enum seg_rw rw;
        unsigned type;
        proc_t *p = ttoproc(ct);
        klwp_t *lwp = ttolwp(ct);
        uintptr_t lofault;
        label_t *onfault;
        faultcode_t pagefault(), res, errcode;
        enum fault_type fault_type;
        k_siginfo_t siginfo;
        uint_t fault = 0;
        int mstate;
        int sicode = 0;
        int watchcode;
        int watchpage;
        caddr_t vaddr;
        int singlestep_twiddle;
        size_t sz;
        int ta;
#ifdef __amd64
        uchar_t instr;
#endif

        ASSERT_STACK_ALIGNED();

        type = rp->r_trapno;
        CPU_STATS_ADDQ(CPU, sys, trap, 1);
        ASSERT(ct->t_schedflag & TS_DONT_SWAP);

        if (type == T_PGFLT) {

                errcode = rp->r_err;
                if (errcode & PF_ERR_WRITE)
                        rw = S_WRITE;
                else if ((caddr_t)rp->r_pc == addr ||
                    (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC)))
                        rw = S_EXEC;
                else
                        rw = S_READ;

#if defined(__i386)
                /*
                 * Pentium Pro work-around
                 */
                if ((errcode & PF_ERR_PROT) && pentiumpro_bug4046376) {
```

```
                        uint_t  attr;
                        uint_t  priv_violation;
                        uint_t  access_violation;

                        if (hat_getattr(addr < (caddr_t)kernelbase ?
                            curproc->p_as->a_hat : kas.a_hat, addr, &attr)
                            == -1) {
                                errcode &= ~PF_ERR_PROT;
                        } else {
                                priv_violation = (errcode & PF_ERR_USER) &&
                                    !(attr & PROT_USER);
                                access_violation = (errcode & PF_ERR_WRITE) &&
                                    !(attr & PROT_WRITE);
                                if (!priv_violation && !access_violation)
                                        goto cleanup;
                        }
                }
#endif /* __i386 */

        } else if (type == T_SGLSTP && lwp != NULL)
                lwp->lwp_pcb.pcb_drstat = (uintptr_t)addr;

        if (tdebug)
                showregs(type, rp, addr);

        if (USERMODE(rp->r_cs)) {
                /*
                 * Set up the current cred to use during this trap. u_cred
                 * no longer exists.  t_cred is used instead.
                 * The current process credential applies to the thread for
                 * the entire trap.  If trapping from the kernel, this
                 * should already be set up.
                 */
                if (ct->t_cred != p->p_cred) {
                        cred_t *oldcred = ct->t_cred;
                        /*
                         * DTrace accesses t_cred in probe context.  t_cred
                         * must always be either NULL, or point to a valid,
                         * allocated cred structure.
                         */
                        ct->t_cred = crgetcred();
                        crfree(oldcred);
                }
                ASSERT(lwp != NULL);
                type |= USER;
                ASSERT(lwptoregs(lwp) == rp);
                lwp->lwp_state = LWP_SYS;

                switch (type) {
                case T_PGFLT + USER:
                        if ((caddr_t)rp->r_pc == addr)
                                mstate = LMS_TFAULT;
                        else
                                mstate = LMS_DFAULT;
                        break;
                default:
                        mstate = LMS_TRAP;
                        break;
                }
                /* Kernel probe */
                TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
                    tnf_microstate, state, mstate);
                mstate = new_mstate(ct, mstate);

                bzero(&siginfo, sizeof (siginfo));
        }
```

```c
        switch (type) {
        case T_PGFLT + USER:
        case T_SGLSTP:
        case T_SGLSTP + USER:
        case T_BPTFLT + USER:
                break;

        default:
                FTRACE_2("trap(): type=0x%lx, regs=0x%lx",
                    (ulong_t)type, (ulong_t)rp);
                break;
        }

        switch (type) {
        case T_SIMDFPE:
                /* Make sure we enable interrupts before die()ing */
                sti();  /* The SIMD exception comes in via cmninttrap */
                /*FALLTHROUGH*/
        default:
                if (type & USER) {
                        if (tudebug)
                                showregs(type, rp, (caddr_t)0);
                        printf("trap: Unknown trap type %d in user mode\n",
                            type & ~USER);
                        siginfo.si_signo = SIGILL;
                        siginfo.si_code  = ILL_ILLTRP;
                        siginfo.si_addr  = (caddr_t)rp->r_pc;
                        siginfo.si_trapno = type & ~USER;
                        fault = FLTILL;
                        break;
                } else {
                        (void) die(type, rp, addr, cpuid);
                        /*NOTREACHED*/
                }

        case T_PGFLT:           /* system page fault */
                /*
                 * If we’re under on_trap() protection (see <sys/ontrap.h>),
                 * set ot_trap and bounce back to the on_trap() call site
                 * via the installed trampoline.
                 */
                if ((ct->t_ontrap != NULL) &&
                    (ct->t_ontrap->ot_prot & OT_DATA_ACCESS)) {
                        ct->t_ontrap->ot_trap |= OT_DATA_ACCESS;
                        rp->r_pc = ct->t_ontrap->ot_trampoline;
                        goto cleanup;
                }

                /*
                 * If we have an Instruction fault in kernel mode, then that
                 * means we’ve tried to execute a user page (SMEP) or both of
                 * PAE and NXE are enabled. In either case, given that it’s a
                 * kernel fault, we should panic immediately and not try to make
                 * any more forward progress. This indicates a bug in the
                 * kernel, which if execution continued, could be exploited to
                 * wreak havoc on the system.
                 */
                if (errcode & PF_ERR_EXEC) {
                        (void) die(type, rp, addr, cpuid);
                }

                /*
                 * See if we can handle as pagefault. Save lofault and onfault
                 * across this. Here we assume that an address less than
                 * KERNELBASE is a user fault.  We can do this as copy.s
```

```c
                 * routines verify that the starting address is less than
                 * KERNELBASE before starting and because we know that we
                 * always have KERNELBASE mapped as invalid to serve as a
                 * "barrier".
                 */
                lofault = ct->t_lofault;
                onfault = ct->t_onfault;
                ct->t_lofault = 0;

                mstate = new_mstate(ct, LMS_KFAULT);

                if (addr < (caddr_t)kernelbase) {
                        res = pagefault(addr,
                            (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 0);
                        if (res == FC_NOMAP &&
                            addr < p->p_usrstack &&
                            grow(addr))
                                res = 0;
                } else {
                        res = pagefault(addr,
                            (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 1);
                }
                (void) new_mstate(ct, mstate);

                /*
                 * Restore lofault and onfault. If we resolved the fault, exit.
                 * If we didn’t and lofault wasn’t set, die.
                 */
                ct->t_lofault = lofault;
                ct->t_onfault = onfault;
                if (res == 0)
                        goto cleanup;

#if defined(OPTERON_ERRATUM_93) && defined(_LP64)
                if (lofault == 0 && opteron_erratum_93) {
                        /*
                         * Workaround for Opteron Erratum 93. On return from
                         * a System Managment Interrupt at a HLT instruction
                         * the %rip might be truncated to a 32 bit value.
                         * BIOS is supposed to fix this, but some don’t.
                         * If this occurs we simply restore the high order bits.
                         * The HLT instruction is 1 byte of 0xf4.
                         */
                        uintptr_t       rip = rp->r_pc;

                        if ((rip & 0xfffffffful) == rip) {
                                rip |= 0xfffffffful << 32;
                                if (hat_getpfnum(kas.a_hat, (caddr_t)rip) !=
                                    PFN_INVALID &&
                                    (*(uchar_t *)rip == 0xf4 ||
                                    *(uchar_t *)(rip - 1) == 0xf4)) {
                                        rp->r_pc = rip;
                                        goto cleanup;
                                }
                        }
                }
#endif /* OPTERON_ERRATUM_93 && _LP64 */

#ifdef OPTERON_ERRATUM_91
                if (lofault == 0 && opteron_erratum_91) {
                        /*
                         * Workaround for Opteron Erratum 91. Prefetches may
                         * generate a page fault (they’re not supposed to do
                         * that!). If this occurs we simply return back to the
                         * instruction.
                         */
                        caddr_t         pc = (caddr_t)rp->r_pc;

                        /*
                         * If the faulting PC is not mapped, this is a
                         * legitimate kernel page fault that must result in a
                         * panic. If the faulting PC is mapped, it could contain
                         * a prefetch instruction. Check for that here.
                         */
                        if (hat_getpfnum(kas.a_hat, pc) != PFN_INVALID) {
                                if (cmp_to_prefetch((uchar_t *)pc)) {
#ifdef DEBUG
                                        cmn_err(CE_WARN, "Opteron erratum 91 "
                                            "occurred: kernel prefetch"
                                            " at %p generated a page fault!",
                                            (void *)rp->r_pc);
#endif /* DEBUG */
                                        goto cleanup;
                                }
                        }
                        (void) die(type, rp, addr, cpuid);
                }
#endif /* OPTERON_ERRATUM_91 */

                if (lofault == 0)
                        (void) die(type, rp, addr, cpuid);

                /*
                 * Cannot resolve fault.  Return to lofault.
                 */
                if (lodebug) {
                        showregs(type, rp, addr);
                        traceregs(rp);
                }
                if (FC_CODE(res) == FC_OBJERR)
                        res = FC_ERRNO(res);
                else
                        res = EFAULT;
                rp->r_r0 = res;
                rp->r_pc = ct->t_lofault;
                goto cleanup;

        case T_PGFLT + USER:    /* user page fault */
                if (faultdebug) {
                        char *fault_str;

                        switch (rw) {
                        case S_READ:
                                fault_str = "read";
                                break;
                        case S_WRITE:
                                fault_str = "write";
                                break;
                        case S_EXEC:
                                fault_str = "exec";
                                break;
                        default:
                                fault_str = "";
                                break;
                        }
                        printf("user %s fault:  addr=0x%lx errcode=0x%x\n",
                            fault_str, (uintptr_t)addr, errcode);
                }

#if defined(OPTERON_ERRATUM_100) && defined(_LP64)
                /*
                 * Workaround for AMD erratum 100
```

```
                 *
                 * A 32-bit process may receive a page fault on a non
                 * 32-bit address by mistake. The range of the faulting
                 * address will be
                 *
                 *      0xffffffff80000000 .. 0xffffffffffffffff or
                 *      0x0000000100000000 .. 0x000000017fffffff
                 *
                 * The fault is always due to an instruction fetch, however
                 * the value of r_pc should be correct (in 32 bit range),
                 * so we ignore the page fault on the bogus address.
                 */
                if (p->p_model == DATAMODEL_ILP32 &&
                    (0xffffffff80000000 <= (uintptr_t)addr ||
                    (0x100000000 <= (uintptr_t)addr &&
                    (uintptr_t)addr <= 0x17fffffff))) {
                        if (!opteron_erratum_100)
                                panic("unexpected erratum #100");
                        if (rp->r_pc <= 0xffffffff)
                                goto out;
                }
#endif /* OPTERON_ERRATUM_100 && _LP64 */

                ASSERT(!(curthread->t_flag & T_WATCHPT));
                watchpage = (pr_watch_active(p) && pr_is_watchpage(addr, rw));
#ifdef __i386
                /*
                 * In 32-bit mode, the lcall (system call) instruction fetches
                 * one word from the stack, at the stack pointer, because of the
                 * way the call gate is constructed.  This is a bogus
                 * read and should not be counted as a read watchpoint.
                 * We work around the problem here by testing to see if
                 * this situation applies and, if so, simply jumping to
                 * the code in locore.s that fields the system call trap.
                 * The registers on the stack are already set up properly
                 * due to the match between the call gate sequence and the
                 * trap gate sequence.  We just have to adjust the pc.
                 */
                if (watchpage && addr == (caddr_t)rp->r_sp &&
                    rw == S_READ && instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
                        extern void watch_syscall(void);

                        rp->r_pc += LCALLSIZE;
                        watch_syscall();        /* never returns */
                        /* NOTREACHED */
                }
#endif /* __i386 */
                vaddr = addr;
                if (!watchpage || (sz = instr_size(rp, &vaddr, rw)) <= 0)
                        fault_type = (errcode & PF_ERR_PROT)? F_PROT: F_INVAL;
                else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
                    sz, NULL, rw)) != 0) {
                        if (ta) {
                                do_watch_step(vaddr, sz, rw,
                                    watchcode, rp->r_pc);
                                fault_type = F_INVAL;
                        } else {
                                bzero(&siginfo, sizeof (siginfo));
                                siginfo.si_signo = SIGTRAP;
                                siginfo.si_code = watchcode;
                                siginfo.si_addr = vaddr;
                                siginfo.si_trapafter = 0;
                                siginfo.si_pc = (caddr_t)rp->r_pc;
                                fault = FLTWATCH;
                                break;
                        }
                } else {
                        /* XXX pr_watch_emul() never succeeds (for now) */
                        if (rw != S_EXEC && pr_watch_emul(rp, vaddr, rw))
                                goto out;
                        do_watch_step(vaddr, sz, rw, 0, 0);
                        fault_type = F_INVAL;
                }

                res = pagefault(addr, fault_type, rw, 0);

                /*
                 * If pagefault() succeeded, ok.
                 * Otherwise attempt to grow the stack.
                 */
                if (res == 0 ||
                    (res == FC_NOMAP &&
                    addr < p->p_usrstack &&
                    grow(addr))) {
                        lwp->lwp_lastfault = FLTPAGE;
                        lwp->lwp_lastfaddr = addr;
                        if (prismember(&p->p_fltmask, FLTPAGE)) {
                                bzero(&siginfo, sizeof (siginfo));
                                siginfo.si_addr = addr;
                                (void) stop_on_fault(FLTPAGE, &siginfo);
                        }
                        goto out;
                } else if (res == FC_PROT && addr < p->p_usrstack &&
                    (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC))) {
                        report_stack_exec(p, addr);
                }

#ifdef OPTERON_ERRATUM_91
                /*
                 * Workaround for Opteron Erratum 91. Prefetches may generate a
                 * page fault (they’re not supposed to do that!). If this
                 * occurs we simply return back to the instruction.
                 *
                 * We rely on copyin to properly fault in the page with r_pc.
                 */
                if (opteron_erratum_91 &&
                    addr != (caddr_t)rp->r_pc &&
                    instr_is_prefetch((caddr_t)rp->r_pc)) {
#ifdef DEBUG
                        cmn_err(CE_WARN, "Opteron erratum 91 occurred: "
                            "prefetch at %p in pid %d generated a trap!",
                            (void *)rp->r_pc, p->p_pid);
#endif /* DEBUG */
                        goto out;
                }
#endif /* OPTERON_ERRATUM_91 */

                if (tudebug)
                        showregs(type, rp, addr);
                /*
                 * In the case where both pagefault and grow fail,
                 * set the code to the value provided by pagefault.
                 * We map all errors returned from pagefault() to SIGSEGV.
                 */
                bzero(&siginfo, sizeof (siginfo));
                siginfo.si_addr = addr;
                switch (FC_CODE(res)) {
                case FC_HWERR:
                case FC_NOSUPPORT:
                        siginfo.si_signo = SIGBUS;
                        siginfo.si_code = BUS_ADRERR;
                        fault = FLTACCESS;
```

```
                        break;
                case FC_ALIGN:
                        siginfo.si_signo = SIGBUS;
                        siginfo.si_code = BUS_ADRALN;
                        fault = FLTACCESS;
                        break;
                case FC_OBJERR:
                        if ((siginfo.si_errno = FC_ERRNO(res)) != EINTR) {
                                siginfo.si_signo = SIGBUS;
                                siginfo.si_code = BUS_OBJERR;
                                fault = FLTACCESS;
                        }
                        break;
                default:        /* FC_NOMAP or FC_PROT */
                        siginfo.si_signo = SIGSEGV;
                        siginfo.si_code =
                            (res == FC_NOMAP)? SEGV_MAPERR : SEGV_ACCERR;
                        fault = FLTBOUNDS;
                        break;
                }
                break;

        case T_ILLINST + USER:  /* invalid opcode fault */
                /*
                 * If the syscall instruction is disabled due to LDT usage, a
                 * user program that attempts to execute it will trigger a #ud
                 * trap. Check for that case here. If this occurs on a CPU which
                 * doesn’t even support syscall, the result of all of this will
                 * be to emulate that particular instruction.
                 */
                if (p->p_ldt != NULL &&
                    ldt_rewrite_syscall(rp, p, X86FSET_ASYSC))
                        goto out;

#ifdef __amd64
                /*
                 * Emulate the LAHF and SAHF instructions if needed.
                 * See the instr_is_lsahf function for details.
                 */
                if (p->p_model == DATAMODEL_LP64 &&
                    instr_is_lsahf((caddr_t)rp->r_pc, &instr)) {
                        emulate_lsahf(rp, instr);
                        goto out;
                }
#endif

                /*FALLTHROUGH*/

                if (tudebug)
                        showregs(type, rp, (caddr_t)0);
                siginfo.si_signo = SIGILL;
                siginfo.si_code  = ILL_ILLOPC;
                siginfo.si_addr  = (caddr_t)rp->r_pc;
                fault = FLTILL;
                break;

        case T_ZERODIV + USER:          /* integer divide by zero */
                if (tudebug && tudebugfpe)
                        showregs(type, rp, (caddr_t)0);
                siginfo.si_signo = SIGFPE;
                siginfo.si_code  = FPE_INTDIV;
                siginfo.si_addr  = (caddr_t)rp->r_pc;
                fault = FLTIZDIV;
                break;

        case T_OVFLW + USER:    /* integer overflow */
```

```
                if (tudebug && tudebugfpe)
                        showregs(type, rp, (caddr_t)0);
                siginfo.si_signo = SIGFPE;
                siginfo.si_code  = FPE_INTOVF;
                siginfo.si_addr  = (caddr_t)rp->r_pc;
                fault = FLTIOVF;
                break;

        case T_NOEXTFLT + USER: /* math coprocessor not available */
                if (tudebug && tudebugfpe)
                        showregs(type, rp, addr);
                if (fpnoextflt(rp)) {
                        siginfo.si_signo = SIGILL;
                        siginfo.si_code  = ILL_ILLOPC;
                        siginfo.si_addr  = (caddr_t)rp->r_pc;
                        fault = FLTILL;
                }
                break;

        case T_EXTOVRFLT:       /* extension overrun fault */
                /* check if we took a kernel trap on behalf of user */
                {
                        extern  void ndptrap_frstor(void);
                        if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
                                sti(); /* T_EXTOVRFLT comes in via cmninttrap */
                                (void) die(type, rp, addr, cpuid);
                        }
                        type |= USER;
                }
                /*FALLTHROUGH*/
        case T_EXTOVRFLT + USER:        /* extension overrun fault */
                if (tudebug && tudebugfpe)
                        showregs(type, rp, addr);
                if (fpextovrflt(rp)) {
                        siginfo.si_signo = SIGSEGV;
                        siginfo.si_code  = SEGV_MAPERR;
                        siginfo.si_addr  = (caddr_t)rp->r_pc;
                        fault = FLTBOUNDS;
                }
                break;

        case T_EXTERRFLT:       /* x87 floating point exception pending */
                /* check if we took a kernel trap on behalf of user */
                {
                        extern  void ndptrap_frstor(void);
                        if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
                                sti(); /* T_EXTERRFLT comes in via cmninttrap */
                                (void) die(type, rp, addr, cpuid);
                        }
                        type |= USER;
                }
                /*FALLTHROUGH*/

        case T_EXTERRFLT + USER: /* x87 floating point exception pending */
                if (tudebug && tudebugfpe)
                        showregs(type, rp, addr);
                if (sicode = fpexterrflt(rp)) {
                        siginfo.si_signo = SIGFPE;
                        siginfo.si_code  = sicode;
                        siginfo.si_addr  = (caddr_t)rp->r_pc;
                        fault = FLTFPE;
                }
                break;

        case T_SIMDFPE + USER:          /* SSE and SSE2 exceptions */
                if (tudebug && tudebugsse)
```

```
                        showregs(type, rp, addr);
                if (!is_x86_feature(x86_featureset, X86FSET_SSE) &&
                    !is_x86_feature(x86_featureset, X86FSET_SSE2)) {
                        /*
                         * There are rumours that some user instructions
                         * on older CPUs can cause this trap to occur; in
                         * which case send a SIGILL instead of a SIGFPE.
                         */
                        siginfo.si_signo = SIGILL;
                        siginfo.si_code  = ILL_ILLTRP;
                        siginfo.si_addr  = (caddr_t)rp->r_pc;
                        siginfo.si_trapno = type & ~USER;
                        fault = FLTILL;
                } else if ((sicode = fpsimderrflt(rp)) != 0) {
                        siginfo.si_signo = SIGFPE;
                        siginfo.si_code = sicode;
                        siginfo.si_addr = (caddr_t)rp->r_pc;
                        fault = FLTFPE;
                }

                sti();  /* The SIMD exception comes in via cmninttrap */
                break;

        case T_BPTFLT:  /* breakpoint trap */
                /*
                 * Kernel breakpoint traps should only happen when kmdb is
                 * active, and even then, it'll have interposed on the IDT, so
                 * control won't get here.  If it does, we've hit a breakpoint
                 * without the debugger, which is very strange, and very
                 * fatal.
                 */
                if (tudebug && tudebugbpt)
                        showregs(type, rp, (caddr_t)0);

                (void) die(type, rp, addr, cpuid);
                break;

        case T_SGLSTP: /* single step/hw breakpoint exception */

                /* Now evaluate how we got here */
                if (lwp != NULL && (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP)) {
                        /*
                         * i386 single-steps even through lcalls which
                         * change the privilege level. So we take a trap at
                         * the first instruction in privileged mode.
                         *
                         * Set a flag to indicate that upon completion of
                         * the system call, deal with the single-step trap.
                         *
                         * The same thing happens for sysenter, too.
                         */
                        singlestep_twiddle = 0;
                        if (rp->r_pc == (uintptr_t)sys_sysenter ||
                            rp->r_pc == (uintptr_t)brand_sys_sysenter) {
                                singlestep_twiddle = 1;
#if defined(__amd64)
                                /*
                                 * Since we are already on the kernel's
                                 * %gs, on 64-bit systems the sysenter case
                                 * needs to adjust the pc to avoid
                                 * executing the swapgs instruction at the
                                 * top of the handler.
                                 */
                                if (rp->r_pc == (uintptr_t)sys_sysenter)
                                        rp->r_pc = (uintptr_t)
                                            _sys_sysenter_post_swapgs;
                                else
                                        rp->r_pc = (uintptr_t)
                                            _brand_sys_sysenter_post_swapgs;
#endif
                        }
#if defined(__i386)
                        else if (rp->r_pc == (uintptr_t)sys_call ||
                            rp->r_pc == (uintptr_t)brand_sys_call) {
                                singlestep_twiddle = 1;
                        }
#endif
                        else {
                                /* not on sysenter/syscall; uregs available */
                                if (tudebug && tudebugbpt)
                                        showregs(type, rp, (caddr_t)0);
                        }
                        if (singlestep_twiddle) {
                                rp->r_ps &= ~PS_T; /* turn off trace */
                                lwp->lwp_pcb.pcb_flags |= DEBUG_PENDING;
                                ct->t_post_sys = 1;
                                aston(curthread);
                                goto cleanup;
                        }
                }
                /* XXX - needs review on debugger interface? */
                if (boothowto & RB_DEBUG)
                        debug_enter((char *)NULL);
                else
                        (void) die(type, rp, addr, cpuid);
                break;

        case T_NMIFLT:  /* NMI interrupt */
                printf("Unexpected NMI in system mode\n");
                goto cleanup;

        case T_NMIFLT + USER:   /* NMI interrupt */
                printf("Unexpected NMI in user mode\n");
                break;

        case T_GPFLT:   /* general protection violation */
                /*
                 * Any #GP that occurs during an on_trap .. no_trap bracket
                 * with OT_DATA_ACCESS or OT_SEGMENT_ACCESS protection,
                 * or in a on_fault .. no_fault bracket, is forgiven
                 * and we trampoline.  This protection is given regardless
                 * of whether we are 32/64 bit etc - if a distinction is
                 * required then define new on_trap protection types.
                 *
                 * On amd64, we can get a #gp from referencing addresses
                 * in the virtual address hole e.g. from a copyin or in
                 * update_sregs while updating user segment registers.
                 *
                 * On the 32-bit hypervisor we could also generate one in
                 * mfn_to_pfn by reaching around or into where the hypervisor
                 * lives which is protected by segmentation.
                 */

                /*
                 * If we're under on_trap() protection (see <sys/ontrap.h>),
                 * set ot_trap and trampoline back to the on_trap() call site
                 * for OT_DATA_ACCESS or OT_SEGMENT_ACCESS.
                 */
                if (ct->t_ontrap != NULL) {
                        int ttype =  ct->t_ontrap->ot_prot &
                            (OT_DATA_ACCESS | OT_SEGMENT_ACCESS);
```

```
                        if (ttype != 0) {
                                ct->t_ontrap->ot_trap |= ttype;
                                if (tudebug)
                                        showregs(type, rp, (caddr_t)0);
                                rp->r_pc = ct->t_ontrap->ot_trampoline;
                                goto cleanup;
                        }
                }

                /*
                 * If we're under lofault protection (copyin etc.),
                 * longjmp back to lofault with an EFAULT.
                 */
                if (ct->t_lofault) {
                        /*
                         * Fault is not resolvable, so just return to lofault
                         */
                        if (lodebug) {
                                showregs(type, rp, addr);
                                traceregs(rp);
                        }
                        rp->r_r0 = EFAULT;
                        rp->r_pc = ct->t_lofault;
                        goto cleanup;
                }

                /*
                 * We fall through to the next case, which repeats
                 * the OT_SEGMENT_ACCESS check which we've already
                 * done, so we'll always fall through to the
                 * T_STKFLT case.
                 */
                /*FALLTHROUGH*/
        case T_SEGFLT:  /* segment not present fault */
                /*
                 * One example of this is #NP in update_sregs while
                 * attempting to update a user segment register
                 * that points to a descriptor that is marked not
                 * present.
                 */
                if (ct->t_ontrap != NULL &&
                    ct->t_ontrap->ot_prot & OT_SEGMENT_ACCESS) {
                        ct->t_ontrap->ot_trap |= OT_SEGMENT_ACCESS;
                        if (tudebug)
                                showregs(type, rp, (caddr_t)0);
                        rp->r_pc = ct->t_ontrap->ot_trampoline;
                        goto cleanup;
                }
                /*FALLTHROUGH*/
        case T_STKFLT:  /* stack fault */
        case T_TSSFLT:  /* invalid TSS fault */
                if (tudebug)
                        showregs(type, rp, (caddr_t)0);
                if (kern_gpfault(rp))
                        (void) die(type, rp, addr, cpuid);
                goto cleanup;

        /*
         * ONLY 32-bit PROCESSES can USE a PRIVATE LDT! 64-bit apps
         * should have no need for them, so we put a stop to it here.
         *
         * So: not-present fault is ONLY valid for 32-bit processes with
         * a private LDT trying to do a system call. Emulate it.
         *
         * #gp fault is ONLY valid for 32-bit processes also, which DO NOT
         * have a private LDT, and are trying to do a system call. Emulate it.
```

```
	 */

	case T_SEGFLT + USER:	/* segment not present fault */
	case T_GPFLT + USER:	/* general protection violation */
#ifdef _SYSCALL32_IMPL
		if (p->p_model != DATAMODEL_NATIVE) {
#endif /* _SYSCALL32_IMPL */
		if (instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
			if (type == T_SEGFLT + USER)
				ASSERT(p->p_ldt != NULL);

			if ((p->p_ldt == NULL && type == T_GPFLT + USER) ||
			    type == T_SEGFLT + USER) {

			/*
			 * The user attempted a system call via the obsolete
			 * call gate mechanism. Because the process doesn't have
			 * an LDT (i.e. the ldtr contains 0), a #gp results.
			 * Emulate the syscall here, just as we do above for a
			 * #np trap.
			 */

			/*
			 * Since this is a not-present trap, rp->r_pc points to
			 * the trapping lcall instruction. We need to bump it
			 * to the next insn so the app can continue on.
			 */
			rp->r_pc += LCALLSIZE;
			lwp->lwp_regs = rp;

			/*
			 * Normally the microstate of the LWP is forced back to
			 * LMS_USER by the syscall handlers. Emulate that
			 * behavior here.
			 */
			mstate = LMS_USER;

			dosyscall();
			goto out;
			}
		}
#ifdef _SYSCALL32_IMPL
		}
#endif /* _SYSCALL32_IMPL */
		/*
		 * If the current process is using a private LDT and the
		 * trapping instruction is sysenter, the sysenter instruction
		 * has been disabled on the CPU because it destroys segment
		 * registers. If this is the case, rewrite the instruction to
		 * be a safe system call and retry it. If this occurs on a CPU
		 * which doesn't even support sysenter, the result of all of
		 * this will be to emulate that particular instruction.
		 */
		if (p->p_ldt != NULL &&
		    ldt_rewrite_syscall(rp, p, X86FSET_SEP))
			goto out;

		/*FALLTHROUGH*/

	case T_BOUNDFLT + USER:	/* bound fault */
	case T_STKFLT + USER:	/* stack fault */
	case T_TSSFLT + USER:	/* invalid TSS fault */
		if (tudebug)
			showregs(type, rp, (caddr_t)0);
		siginfo.si_signo = SIGSEGV;
		siginfo.si_code  = SEGV_MAPERR;
```

```
		siginfo.si_addr  = (caddr_t)rp->r_pc;
		fault = FLTBOUNDS;
		break;

	case T_ALIGNMENT + USER:	/* user alignment error (486) */
		if (tudebug)
			showregs(type, rp, (caddr_t)0);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_signo = SIGBUS;
		siginfo.si_code = BUS_ADRALN;
		siginfo.si_addr = (caddr_t)rp->r_pc;
		fault = FLTACCESS;
		break;

	case T_SGLSTP + USER: /* single step/hw breakpoint exception */
		if (tudebug && tudebugbpt)
			showregs(type, rp, (caddr_t)0);

		/* Was it single-stepping? */
		if (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP) {
			pcb_t *pcb = &lwp->lwp_pcb;

			rp->r_ps &= ~PS_T;
			/*
			 * If both NORMAL_STEP and WATCH_STEP are in effect,
			 * give precedence to WATCH_STEP.  If neither is set,
			 * user must have set the PS_T bit in %efl; treat this
			 * as NORMAL_STEP.
			 */
			if ((fault = undo_watch_step(&siginfo)) == 0 &&
			    ((pcb->pcb_flags & NORMAL_STEP) ||
			    !(pcb->pcb_flags & WATCH_STEP))) {
				siginfo.si_signo = SIGTRAP;
				siginfo.si_code = TRAP_TRACE;
				siginfo.si_addr = (caddr_t)rp->r_pc;
				fault = FLTTRACE;
			}
			pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
		}
		break;

	case T_BPTFLT + USER:	/* breakpoint trap */
		if (tudebug && tudebugbpt)
			showregs(type, rp, (caddr_t)0);
		/*
		 * int 3 (the breakpoint instruction) leaves the pc referring
		 * to the address one byte after the breakpointed address.
		 * If the P_PR_BPTADJ flag has been set via /proc, We adjust
		 * it back so it refers to the breakpointed address.
		 */
		if (p->p_proc_flag & P_PR_BPTADJ)
			rp->r_pc--;
		siginfo.si_signo = SIGTRAP;
		siginfo.si_code  = TRAP_BRKPT;
		siginfo.si_addr  = (caddr_t)rp->r_pc;
		fault = FLTBPT;
		break;

	case T_AST:
		/*
		 * This occurs only after the cs register has been made to
		 * look like a kernel selector, either through debugging or
		 * possibly by functions like setcontext().  The thread is
		 * about to cause a general protection fault at common_iret()
		 * in locore.  We let that happen immediately instead of
		 * doing the T_AST processing.
		 */
		goto cleanup;

	case T_AST + USER:	/* profiling, resched, h/w error pseudo trap */
		if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
			proc_t *p = ttoproc(curthread);
			extern void print_msg_hwerr(ctid_t ct_id, proc_t *p);

			lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
			print_msg_hwerr(p->p_ct_process->conp_contract.ct_id,
			    p);
			contract_process_hwerr(p->p_ct_process, p);
			siginfo.si_signo = SIGKILL;
			siginfo.si_code = SI_NOINFO;
		} else if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
			lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
			if (kcpc_overflow_ast()) {
				/*
				 * Signal performance counter overflow
				 */
				if (tudebug)
					showregs(type, rp, (caddr_t)0);
				bzero(&siginfo, sizeof (siginfo));
				siginfo.si_signo = SIGEMT;
				siginfo.si_code = EMT_CPCOVF;
				siginfo.si_addr = (caddr_t)rp->r_pc;
				fault = FLTCPCOVF;
			}
		}

		break;
	}

	/*
	 * We can't get here from a system trap
	 */
	ASSERT(type & USER);

	if (fault) {
		/* We took a fault so abort single step. */
		lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
		/*
		 * Remember the fault and fault adddress
		 * for real-time (SIGPROF) profiling.
		 */
		lwp->lwp_lastfault = fault;
		lwp->lwp_lastfaddr = siginfo.si_addr;

		DTRACE_PROC2(fault, int, fault, ksiginfo_t *, &siginfo);

		/*
		 * If a debugger has declared this fault to be an
		 * event of interest, stop the lwp.  Otherwise just
		 * deliver the associated signal.
		 */
		if (siginfo.si_signo != SIGKILL &&
		    prismember(&p->p_fltmask, fault) &&
		    stop_on_fault(fault, &siginfo) == 0)
			siginfo.si_signo = 0;
	}

	if (siginfo.si_signo)
		trapsig(&siginfo, (fault != FLTFPE && fault != FLTCPCOVF));

	if (lwp->lwp_oweupc)
		profil_tick(rp->r_pc);
```

```

	if (ct->t_astflag | ct->t_sig_check) {
		/*
		 * Turn off the AST flag before checking all the conditions that
		 * may have caused an AST.  This flag is on whenever a signal or
		 * unusual condition should be handled after the next trap or
		 * syscall.
		 */
		astoff(ct);
		/*
		 * If a single-step trap occurred on a syscall (see above)
		 * recognize it now.  Do this before checking for signals
		 * because deferred_singlestep_trap() may generate a SIGTRAP to
		 * the LWP or may otherwise mark the LWP to call issig(FORREAL).
		 */
		if (lwp->lwp_pcb.pcb_flags & DEBUG_PENDING)
			deferred_singlestep_trap((caddr_t)rp->r_pc);

		ct->t_sig_check = 0;

		mutex_enter(&p->p_lock);
		if (curthread->t_proc_flag & TP_CHANGEBIND) {
			timer_lwpbind();
			curthread->t_proc_flag &= ~TP_CHANGEBIND;
		}
		mutex_exit(&p->p_lock);

		/*
		 * for kaio requests that are on the per-process poll queue,
		 * aiop->aio_pollq, they're AIO_POLL bit is set, the kernel
		 * should copyout their result_t to user memory. by copying
		 * out the result_t, the user can poll on memory waiting
		 * for the kaio request to complete.
		 */
		if (p->p_aio)
			aio_cleanup(0);
		/*
		 * If this LWP was asked to hold, call holdlwp(), which will
		 * stop.  holdlwps() sets this up and calls pokelwps() which
		 * sets the AST flag.
		 *
		 * Also check TP_EXITLWP, since this is used by fresh new LWPs
		 * through lwp_rtt().  That flag is set if the lwp_create(2)
		 * syscall failed after creating the LWP.
		 */
		if (ISHOLD(p))
			holdlwp();

		/*
		 * All code that sets signals and makes ISSIG evaluate true must
		 * set t_astflag afterwards.
		 */
		if (ISSIG_PENDING(ct, lwp, p)) {
			if (issig(FORREAL))
				psig();
			ct->t_sig_check = 1;
		}

		if (ct->t_rprof != NULL) {
			realsigprof(0, 0, 0);
			ct->t_sig_check = 1;
		}

		/*
		 * /proc can't enable/disable the trace bit itself
		 * because that could race with the call gate used by
```

```
                 * system calls via "lcall". If that happened, an
                 * invalid EFLAGS would result. prstep()/prnostep()
                 * therefore schedule an AST for the purpose.
                 */
                if (lwp->lwp_pcb.pcb_flags & REQUEST_STEP) {
                        lwp->lwp_pcb.pcb_flags &= ~REQUEST_STEP;
                        rp->r_ps |= PS_T;
                }
                if (lwp->lwp_pcb.pcb_flags & REQUEST_NOSTEP) {
                        lwp->lwp_pcb.pcb_flags &= ~REQUEST_NOSTEP;
                        rp->r_ps &= ~PS_T;
                }
        }

out:    /* We can't get here from a system trap */
        ASSERT(type & USER);

        if (ISHOLD(p))
                holdlwp();

        /*
         * Set state to LWP_USER here so preempt won't give us a kernel
         * priority if it occurs after this point.  Call CL_TRAPRET() to
         * restore the user-level priority.
         *
         * It is important that no locks (other than spinlocks) be entered
         * after this point before returning to user mode (unless lwp_state
         * is set back to LWP_SYS).
         */
        lwp->lwp_state = LWP_USER;

        if (ct->t_trapret) {
                ct->t_trapret = 0;
                thread_lock(ct);
                CL_TRAPRET(ct);
                thread_unlock(ct);
        }
        if (CPU->cpu_runrun || curthread->t_schedflag & TS_ANYWAITQ)
                preempt();
        prunstop();
        (void) new_mstate(ct, mstate);

        /* Kernel probe */
        TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
            tnf_microstate, state, LMS_USER);

        return;

cleanup:        /* system traps end up here */
        ASSERT(!(type & USER));
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
  105436 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/vm/hat_i86.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*
 * Allocate any hat resources required for a process being swapped in.
 */
/*ARGSUSED*/
void
hat_swapin(hat_t *hat)
{
        /* do nothing - we let everything fault back in */
}

/*
 * Unload all translations associated with an address space of a process
 * that is being swapped out.
 */
void
hat_swapout(hat_t *hat)
{
        uintptr_t       vaddr = (uintptr_t)0;
        uintptr_t       eaddr = _userlimit;
        htable_t        *ht = NULL;
        level_t         l;

        XPV_DISALLOW_MIGRATE();
        /*
         * We can't just call hat_unload(hat, 0, _userlimit...)  here, because
         * seg_spt and shared pagetables can't be swapped out.
         * Take a look at segspt_shmswapout() - it's a big no-op.
         *
         * Instead we'll walk through all the address space and unload
         * any mappings which we are sure are not shared, not locked.
         */
        ASSERT(IS_PAGEALIGNED(vaddr));
        ASSERT(IS_PAGEALIGNED(eaddr));
        ASSERT(AS_LOCK_HELD(hat->hat_as));
        if ((uintptr_t)hat->hat_as->a_userlimit < eaddr)
                eaddr = (uintptr_t)hat->hat_as->a_userlimit;

        while (vaddr < eaddr) {
                (void) htable_walk(hat, &ht, &vaddr, eaddr);
                if (ht == NULL)
                        break;

                ASSERT(!IN_VA_HOLE(vaddr));

                /*
                 * If the page table is shared skip its entire range.
                 */
                l = ht->ht_level;
                if (ht->ht_flags & HTABLE_SHARED_PFN) {
                        vaddr = ht->ht_vaddr + LEVEL_SIZE(l + 1);
                        htable_release(ht);
                        ht = NULL;
                        continue;
                }

                /*
                 * If the page table has no locked entries, unload this one.
                 */
                if (ht->ht_lock_cnt == 0)
```

```
                        hat_unload(hat, (caddr_t)vaddr, LEVEL_SIZE(l),
                            HAT_UNLOAD_UNMAP);

                /*
                 * If we have a level 0 page table with locked entries,
                 * skip the entire page table, otherwise skip just one entry.
                 */
                if (ht->ht_lock_cnt > 0 && l == 0)
                        vaddr = ht->ht_vaddr + LEVEL_SIZE(1);
                else
                        vaddr += LEVEL_SIZE(l);
        }
        if (ht)
                htable_release(ht);

        /*
         * We're in swapout because the system is low on memory, so
         * go back and flush all the htables off the cached list.
         */
        htable_purge_hat(hat);
        XPV_ALLOW_MIGRATE();
}

/*
 * returns number of bytes that have valid mappings in hat.
 */
size_t
hat_get_mapped_size(hat_t *hat)
{
        size_t total = 0;
        int l;

        for (l = 0; l <= mmu.max_page_level; l++)
                total += (hat->hat_pages_mapped[l] << LEVEL_SHIFT(l));
        total += hat->hat_ism_pgcnt;

        return (total);
}
______unchanged_portion_omitted_
```

****
16753 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86xpv/vm/seg_mf.c
6583 remove whole-process swapping
****
______unchanged_portion_omitted_

```
static struct seg_ops segmf_ops = {
        segmf_dup,
        segmf_unmap,
        segmf_free,
        segmf_fault,
        segmf_faulta,
        segmf_setprot,
        segmf_checkprot,
        (int (*)())segmf_kluster,
        (size_t (*)(struct seg *))NULL, /* swapout */
        segmf_sync,
        segmf_incore,
        segmf_lockop,
        segmf_getprot,
        segmf_getoffset,
        segmf_gettype,
        segmf_getvp,
        segmf_advise,
        segmf_dump,
        segmf_pagelock,
        segmf_setpagesize,
        segmf_getmemid,
        segmf_getpolicy,
        segmf_capable,
        seg_inherit_notsup
};
```
______unchanged_portion_omitted_

****
   35882 Thu Feb 18 08:40:44 2016
new/usr/src/uts/intel/ia32/os/syscall.c
6583 remove whole-process swapping
****
______*unchanged_portion_omitted_*

```
/*
 * Called from syscall() when a non-trivial 32-bit system call occurs.
 *      Sets up the args and returns a pointer to the handler.
 */
struct sysent *
syscall_entry(kthread_t *t, long *argp)
{
        klwp_t *lwp = ttolwp(t);
        struct regs *rp = lwptoregs(lwp);
        unsigned int code;
        struct sysent *callp;
        struct sysent *se = LWP_GETSYSENT(lwp);
        int error = 0;
        uint_t nargs;

        ASSERT(t == curthread);
        ASSERT(t == curthread && curthread->t_schedflag & TS_DONT_SWAP);

        lwp->lwp_ru.sysc++;
        lwp->lwp_eosys = NORMALRETURN;  /* assume this will be normal */

        /*
         * Set lwp_ap to point to the args, even if none are needed for this
         * system call.  This is for the loadable-syscall case where the
         * number of args won't be known until the system call is loaded, and
         * also maintains a non-NULL lwp_ap setup for get_syscall_args(). Note
         * that lwp_ap MUST be set to a non-NULL value _BEFORE_ t_sysnum is
         * set to non-zero; otherwise get_syscall_args(), seeing a non-zero
         * t_sysnum for this thread, will charge ahead and dereference lwp_ap.
         */
        lwp->lwp_ap = argp;             /* for get_syscall_args */

        code = rp->r_r0;
        t->t_sysnum = (short)code;
        callp = code >= NSYSCALL ? &nosys_ent : se + code;

        if ((t->t_pre_sys | syscalltrace) != 0) {
                error = pre_syscall();

                /*
                 * pre_syscall() has taken care so that lwp_ap is current;
                 * it either points to syscall-entry-saved amd64 regs,
                 * or it points to lwp_arg[], which has been re-copied from
                 * the ia32 ustack, but either way, it's a current copy after
                 * /proc has possibly mucked with the syscall args.
                 */

                if (error)
                        return (&sysent_err);   /* use dummy handler */
        }

        /*
         * Fetch the system call arguments to the kernel stack copy used
         * for syscall handling.
         * Note: for loadable system calls the number of arguments required
         * may not be known at this point, and will be zero if the system call
         * was never loaded.  Once the system call has been loaded, the number
         * of args is not allowed to be changed.
         */
```

```
        if ((nargs = (uint_t)callp->sy_narg) != 0 &&
            COPYIN_ARGS32(rp, argp, nargs)) {
                (void) set_errno(EFAULT);
                return (&sysent_err);   /* use dummy handler */
        }

        return (callp);         /* return sysent entry for caller */
}
```

______*unchanged_portion_omitted_*

```
/*
 * Perform pre-system-call processing, including stopping for tracing,
 * auditing, etc.
 *
 * This routine is called only if the t_pre_sys flag is set. Any condition
 * requiring pre-syscall handling must set the t_pre_sys flag. If the
 * condition is persistent, this routine will repost t_pre_sys.
 */
int
pre_syscall()
{
        kthread_t *t = curthread;
        unsigned code = t->t_sysnum;
        klwp_t *lwp = ttolwp(t);
        proc_t *p = ttoproc(t);
        int     repost;

        t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

        ASSERT(t->t_schedflag & TS_DONT_SWAP);

#if defined(DEBUG)
        /*
         * On the i386 kernel, lwp_ap points at the piece of the thread
         * stack that we copy the users arguments into.
         *
         * On the amd64 kernel, the syscall arguments in the rdi..r9
         * registers should be pointed at by lwp_ap.  If the args need to
         * be copied so that those registers can be changed without losing
         * the ability to get the args for /proc, they can be saved by
         * save_syscall_args(), and lwp_ap will be restored by post_syscall().
         */
        if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
#if defined(_LP64)
                ASSERT(lwp->lwp_ap == (long *)&lwptoregs(lwp)->r_rdi);
        } else {
#endif
                ASSERT((caddr_t)lwp->lwp_ap > t->t_stkbase &&
                    (caddr_t)lwp->lwp_ap < t->t_stk);
        }
#endif  /* DEBUG */

        /*
         * Make sure the thread is holding the latest credentials for the
         * process.  The credentials in the process right now apply to this
         * thread for the entire system call.
         */
        if (t->t_cred != p->p_cred) {
                cred_t *oldcred = t->t_cred;
                /*
                 * DTrace accesses t_cred in probe context.  t_cred must
                 * always be either NULL, or point to a valid, allocated cred
                 * structure.
                 */
                t->t_cred = crgetcred();
                crfree(oldcred);
        }

        /*
         * From the proc(4) manual page:
         * When entry to a system call is being traced, the traced process
         * stops after having begun the call to the system but before the
         * system call arguments have been fetched from the process.
         */
        if (PTOU(p)->u_systrap) {
                if (prismember(&PTOU(p)->u_entrymask, code)) {
                        mutex_enter(&p->p_lock);
                        /*
                         * Recheck stop condition, now that lock is held.
                         */
                        if (PTOU(p)->u_systrap &&
                            prismember(&PTOU(p)->u_entrymask, code)) {
                                stop(PR_SYSENTRY, code);

                                /*
                                 * /proc may have modified syscall args,
                                 * either in regs for amd64 or on ustack
                                 * for ia32.  Either way, arrange to
                                 * copy them again, both for the syscall
                                 * handler and for other consumers in
                                 * post_syscall (like audit).  Here, we
                                 * only do amd64, and just set lwp_ap
                                 * back to the kernel-entry stack copy;
                                 * the syscall ml code redoes
                                 * move-from-regs to set up for the
                                 * syscall handler after we return.  For
                                 * ia32, save_syscall_args() below makes
                                 * an lwp_ap-accessible copy.
                                 */
#if defined(_LP64)
                                if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
                                        lwp->lwp_argsaved = 0;
                                        lwp->lwp_ap =
                                            (long *)&lwptoregs(lwp)->r_rdi;
                                }
#endif
                        }
                        mutex_exit(&p->p_lock);
                }
                repost = 1;
        }

        /*
         * ia32 kernel, or ia32 proc on amd64 kernel: keep args in
         * lwp_arg for post-syscall processing, regardless of whether
         * they might have been changed in /proc above.
         */
#if defined(_LP64)
        if (lwp_getdatamodel(lwp) != DATAMODEL_NATIVE)
#endif
                (void) save_syscall_args();

        if (lwp->lwp_sysabort) {
                /*
                 * lwp_sysabort may have been set via /proc while the process
                 * was stopped on PR_SYSENTRY.  If so, abort the system call.
                 * Override any error from the copyin() of the arguments.
                 */
                lwp->lwp_sysabort = 0;
                (void) set_errno(EINTR);        /* forces post_sys */
                t->t_pre_sys = 1;       /* repost anyway */
                return (1);             /* don't do system call, return EINTR */
```

```
        }

        /*
         * begin auditing for this syscall if the c2audit module is loaded
         * and auditing is enabled
         */
        if (audit_active == C2AUDIT_LOADED) {
                uint32_t auditing = au_zone_getstate(NULL);

                if (auditing & AU_AUDIT_MASK) {
                        int error;
                        if (error = audit_start(T_SYSCALL, code, auditing, \
                            0, lwp)) {
                                t->t_pre_sys = 1;       /* repost anyway */
                                (void) set_errno(error);
                                return (1);
                        }
                        repost = 1;
                }
        }

#ifndef NPROBE
        /* Kernel probe */
        if (tnf_tracing_active) {
                TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
                        tnf_sysnum,     sysnum,         t->t_sysnum);
                t->t_post_sys = 1;      /* make sure post_syscall runs */
                repost = 1;
        }
#endif /* NPROBE */

#ifdef SYSCALLTRACE
        if (syscalltrace) {
                int i;
                long *ap;
                char *cp;
                char *sysname;
                struct sysent *callp;

                if (code >= NSYSCALL)
                        callp = &nosys_ent;     /* nosys has no args */
                else
                        callp = LWP_GETSYSENT(lwp) + code;
                (void) save_syscall_args();
                mutex_enter(&systrace_lock);
                printf("%d: ", p->p_pid);
                if (code >= NSYSCALL)
                        printf("0x%x", code);
                else {
                        sysname = mod_getsysname(code);
                        printf("%s[0x%x/0x%p]", sysname == NULL ? "NULL" :
                            sysname, code, callp->sy_callc);
                }
                cp = "(";
                for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
                        printf("%s%lx", cp, *ap);
                        cp = ", ";
                }
                if (i)
                        printf(")");
                printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
                mutex_exit(&systrace_lock);
        }
#endif /* SYSCALLTRACE */

        /*
```

```
	 * If there was a continuing reason for pre-syscall processing,
	 * set the t_pre_sys flag for the next system call.
	 */
	if (repost)
		t->t_pre_sys = 1;
	lwp->lwp_error = 0;	/* for old drivers */
	lwp->lwp_badpriv = PRIV_NONE;
	return (0);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
  413119 Thu Feb 18 08:40:44 2016
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

/*
 * Set up any translation structures, for the specified address space,
 * that are needed or preferred when the process is being swapped in.
 */
/* ARGSUSED */
void
hat_swapin(struct hat *hat)
{
}

/*
 * Free all of the translation resources, for the specified address space,
 * that can be freed while the process is swapped out. Called from as_swapout.
 * Also, free up the ctx that this process was using.
 */
void
hat_swapout(struct hat *sfmmup)
{
        struct hmehash_bucket *hmebp;
        struct hme_blk *hmeblkp;
        struct hme_blk *pr_hblk = NULL;
        struct hme_blk *nx_hblk;
        int i;
        struct hme_blk *list = NULL;
        hatlock_t *hatlockp;
        struct tsb_info *tsbinfop;
        struct free_tsb {
                struct free_tsb *next;
                struct tsb_info *tsbinfop;
        };                      /* free list of TSBs */
        struct free_tsb *freelist, *last, *next;

        SFMMU_STAT(sf_swapout);

        /*
         * There is no way to go from an as to all its translations in sfmmu.
         * Here is one of the times when we take the big hit and traverse
         * the hash looking for hme_blks to free up.  Not only do we free up
         * this as hme_blks but all those that are free.  We are obviously
         * swapping because we need memory so let's free up as much
         * as we can.
         *
         * Note that we don't flush TLB/TSB here -- it's not necessary
         * because:
         *  1) we free the ctx we're using and throw away the TSB(s);
         *  2) processes aren't runnable while being swapped out.
         */
        ASSERT(sfmmup != KHATID);
        for (i = 0; i <= UHMEHASH_SZ; i++) {
                hmebp = &uhme_hash[i];
                SFMMU_HASH_LOCK(hmebp);
                hmeblkp = hmebp->hmeblkp;
                pr_hblk = NULL;
                while (hmeblkp) {

                        if ((hmeblkp->hblk_tag.htag_id == sfmmup) &&
                            !hmeblkp->hblk_shw_bit && !hmeblkp->hblk_lckcnt) {
                                ASSERT(!hmeblkp->hblk_shared);
                                (void) sfmmu_hblk_unload(sfmmup, hmeblkp,
```

```
                                    (caddr_t)get_hblk_base(hmeblkp),
                                    get_hblk_endaddr(hmeblkp),
                                    NULL, HAT_UNLOAD);
                        }
                        nx_hblk = hmeblkp->hblk_next;
                        if (!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
                                ASSERT(!hmeblkp->hblk_lckcnt);
                                sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
                                    &list, 0);
                        } else {
                                pr_hblk = hmeblkp;
                        }
                        hmeblkp = nx_hblk;
                }
                SFMMU_HASH_UNLOCK(hmebp);
        }

        sfmmu_hblks_list_purge(&list, 0);

        /*
         * Now free up the ctx so that others can reuse it.
         */
        hatlockp = sfmmu_hat_enter(sfmmup);

        sfmmu_invalidate_ctx(sfmmup);

        /*
         * Free TSBs, but not tsbinfos, and set SWAPPED flag.
         * If TSBs were never swapped in, just return.
         * This implies that we don't support partial swapping
         * of TSBs -- either all are swapped out, or none are.
         *
         * We must hold the HAT lock here to prevent racing with another
         * thread trying to unmap TTEs from the TSB or running the post-
         * relocator after relocating the TSB's memory.  Unfortunately, we
         * can't free memory while holding the HAT lock or we could
         * deadlock, so we build a list of TSBs to be freed after marking
         * the tsbinfos as swapped out and free them after dropping the
         * lock.
         */
        if (SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
                sfmmu_hat_exit(hatlockp);
                return;
        }

        SFMMU_FLAGS_SET(sfmmup, HAT_SWAPPED);
        last = freelist = NULL;
        for (tsbinfop = sfmmup->sfmmu_tsb; tsbinfop != NULL;
            tsbinfop = tsbinfop->tsb_next) {
                ASSERT((tsbinfop->tsb_flags & TSB_SWAPPED) == 0);

                /*
                 * Cast the TSB into a struct free_tsb and put it on the free
                 * list.
                 */
                if (freelist == NULL) {
                        last = freelist = (struct free_tsb *)tsbinfop->tsb_va;
                } else {
                        last->next = (struct free_tsb *)tsbinfop->tsb_va;
                        last = last->next;
                }
                last->next = NULL;
                last->tsbinfop = tsbinfop;
                tsbinfop->tsb_flags |= TSB_SWAPPED;
                /*
                 * Zero out the TTE to clear the valid bit.
                 * Note we can't use a value like 0xbad because we want to
                 * ensure diagnostic bits are NEVER set on TTEs that might
                 * be loaded.  The intent is to catch any invalid access
                 * to the swapped TSB, such as a thread running with a valid
                 * context without first calling sfmmu_tsb_swapin() to
                 * allocate TSB memory.
                 */
                tsbinfop->tsb_tte.ll = 0;
        }

        /* Now we can drop the lock and free the TSB memory. */
        sfmmu_hat_exit(hatlockp);
        for (; freelist != NULL; freelist = next) {
                next = freelist->next;
                sfmmu_tsb_free(freelist->tsbinfop);
        }
}

/*
 * Duplicate the translations of an as into another newas
 */
/* ARGSUSED */
int
hat_dup(struct hat *hat, struct hat *newhat, caddr_t addr, size_t len,
        uint_t flag)
{
        sf_srd_t *srdp;
        sf_scd_t *scdp;
        int i;
        extern uint_t get_color_start(struct as *);

        ASSERT((flag == 0) || (flag == HAT_DUP_ALL) || (flag == HAT_DUP_COW) ||
            (flag == HAT_DUP_SRD));
        ASSERT(hat != ksfmmup);
        ASSERT(newhat != ksfmmup);
        ASSERT(flag != HAT_DUP_ALL || hat->sfmmu_srdp == newhat->sfmmu_srdp);

        if (flag == HAT_DUP_COW) {
                panic("hat_dup: HAT_DUP_COW not supported");
        }

        if (flag == HAT_DUP_SRD && ((srdp = hat->sfmmu_srdp) != NULL)) {
                ASSERT(srdp->srd_evp != NULL);
                VN_HOLD(srdp->srd_evp);
                ASSERT(srdp->srd_refcnt > 0);
                newhat->sfmmu_srdp = srdp;
                atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
        }

        /*
         * HAT_DUP_ALL flag is used after as duplication is done.
         */
        if (flag == HAT_DUP_ALL && ((srdp = newhat->sfmmu_srdp) != NULL)) {
                ASSERT(newhat->sfmmu_srdp->srd_refcnt >= 2);
                newhat->sfmmu_rtteflags = hat->sfmmu_rtteflags;
                if (hat->sfmmu_flags & HAT_4MTEXT_FLAG) {
                        newhat->sfmmu_flags |= HAT_4MTEXT_FLAG;
                }

                /* check if need to join scd */
                if ((scdp = hat->sfmmu_scdp) != NULL &&
                    newhat->sfmmu_scdp != scdp) {
                        int ret;
                        SF_RGNMAP_IS_SUBSET(&newhat->sfmmu_region_map,
                            &scdp->scd_region_map, ret);
                        ASSERT(ret);
```

```
                        sfmmu_join_scd(scdp, newhat);
                        ASSERT(newhat->sfmmu_scdp == scdp &&
                            scdp->scd_refcnt >= 2);
                        for (i = 0; i < max_mmu_page_sizes; i++) {
                                newhat->sfmmu_ismttecnt[i] =
                                    hat->sfmmu_ismttecnt[i];
                                newhat->sfmmu_scdismttecnt[i] =
                                    hat->sfmmu_scdismttecnt[i];
                        }
                }

                sfmmu_check_page_sizes(newhat, 1);
        }

        if (flag == HAT_DUP_ALL && consistent_coloring == 0 &&
            update_proc_pgcolorbase_after_fork != 0) {
                hat->sfmmu_clrbin = get_color_start(hat->sfmmu_as);
        }
        return (0);
}
______unchanged_portion_omitted_


/*
 * Replace the specified TSB with a new TSB.  This function gets called when
 * we grow, or shrink a TSB.  When swapping in a TSB (TSB_SWAPIN), the
 * we grow, shrink or swapin a TSB.  When swapping in a TSB (TSB_SWAPIN), the
 * TSB_FORCEALLOC flag may be used to force allocation of a minimum-sized TSB
 * (8K).
 *
 * Caller must hold the HAT lock, but should assume any tsb_info
 * pointers it has are no longer valid after calling this function.
 *
 * Return values:
 *      TSB_ALLOCFAIL   Failed to allocate a TSB, due to memory constraints
 *      TSB_LOSTRACE    HAT is busy, i.e. another thread is already doing
 *                      something to this tsbinfo/TSB
 *      TSB_SUCCESS     Operation succeeded
 */
static tsb_replace_rc_t
sfmmu_replace_tsb(sfmmu_t *sfmmup, struct tsb_info *old_tsbinfo, uint_t szc,
    hatlock_t *hatlockp, uint_t flags)
{
        struct tsb_info *new_tsbinfo = NULL;
        struct tsb_info *curtsb, *prevtsb;
        uint_t tte_sz_mask;
        int i;

        ASSERT(sfmmup != ksfmmup);
        ASSERT(sfmmup->sfmmu_ismhat == 0);
        ASSERT(sfmmu_hat_lock_held(sfmmup));
        ASSERT(szc <= tsb_max_growsize);

        if (SFMMU_FLAGS_ISSET(sfmmup, HAT_BUSY))
                return (TSB_LOSTRACE);

        /*
         * Find the tsb_info ahead of this one in the list, and
         * also make sure that the tsb_info passed in really
         * exists!
         */
        for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
            curtsb != old_tsbinfo && curtsb != NULL;
            prevtsb = curtsb, curtsb = curtsb->tsb_next)
                ;
        ASSERT(curtsb != NULL);
	if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
		/*
		 * The process is swapped out, so just set the new size
		 * code.  When it swaps back in, we'll allocate a new one
		 * of the new chosen size.
		 */
		curtsb->tsb_szc = szc;
		return (TSB_SUCCESS);
	}
	SFMMU_FLAGS_SET(sfmmup, HAT_BUSY);

	tte_sz_mask = old_tsbinfo->tsb_ttesz_mask;

	/*
	 * All initialization is done inside of sfmmu_tsbinfo_alloc().
	 * If we fail to allocate a TSB, exit.
	 *
	 * If tsb grows with new tsb size > 4M and old tsb size < 4M,
	 * then try 4M slab after the initial alloc fails.
	 *
	 * If tsb swapin with tsb size > 4M, then try 4M after the
	 * initial alloc fails.
	 */
	sfmmu_hat_exit(hatlockp);
	if (sfmmu_tsbinfo_alloc(&new_tsbinfo, szc,
	    tte_sz_mask, flags, sfmmup) &&
	    (!(flags & (TSB_GROW | TSB_SWAPIN)) || (szc <= TSB_4M_SZCODE) ||
	    (!(flags & TSB_SWAPIN) &&
	    (old_tsbinfo->tsb_szc >= TSB_4M_SZCODE)) ||
	    sfmmu_tsbinfo_alloc(&new_tsbinfo, TSB_4M_SZCODE,
	    tte_sz_mask, flags, sfmmup))) {
		(void) sfmmu_hat_enter(sfmmup);
		if (!(flags & TSB_SWAPIN))
			SFMMU_STAT(sf_tsb_resize_failures);
		SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
		return (TSB_ALLOCFAIL);
	}
	(void) sfmmu_hat_enter(sfmmup);

	/*
	 * Re-check to make sure somebody else didn't muck with us while we
	 * didn't hold the HAT lock.  If the process swapped out, fine, just
	 * exit; this can happen if we try to shrink the TSB from the context
	 * of another process (such as on an ISM unmap), though it is rare.
	 */
	if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
		SFMMU_STAT(sf_tsb_resize_failures);
		SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
		sfmmu_hat_exit(hatlockp);
		sfmmu_tsbinfo_free(new_tsbinfo);
		(void) sfmmu_hat_enter(sfmmup);
		return (TSB_LOSTRACE);
	}

#ifdef	DEBUG
	/* Reverify that the tsb_info still exists.. for debugging only */
	for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
	    curtsb != old_tsbinfo && curtsb != NULL;
	    prevtsb = curtsb, curtsb = curtsb->tsb_next)
		;
	ASSERT(curtsb != NULL);
#endif	/* DEBUG */

	/*
	 * Quiesce any CPUs running this process on their next TLB miss
```

```
	 * so they atomically see the new tsb_info.  We temporarily set the
	 * context to invalid context so new threads that come on processor
	 * after we do the xcall to cpusran will also serialize behind the
	 * HAT lock on TLB miss and will see the new TSB.  Since this short
	 * race with a new thread coming on processor is relatively rare,
	 * this synchronization mechanism should be cheaper than always
	 * pausing all CPUs for the duration of the setup, which is what
	 * the old implementation did.  This is particuarly true if we are
	 * copying a huge chunk of memory around during that window.
	 *
	 * The memory barriers are to make sure things stay consistent
	 * with resume() since it does not hold the HAT lock while
	 * walking the list of tsb_info structures.
	 */
	if ((flags & TSB_SWAPIN) != TSB_SWAPIN) {
		/* The TSB is either growing or shrinking. */
		sfmmu_invalidate_ctx(sfmmup);
	} else {
		/*
		 * It is illegal to swap in TSBs from a process other
		 * than a process being swapped in.  This in turn
		 * implies we do not have a valid MMU context here
		 * since a process needs one to resolve translation
		 * misses.
		 */
		ASSERT(curthread->t_procp->p_as->a_hat == sfmmup);
	}

#ifdef DEBUG
	ASSERT(max_mmu_ctxdoms > 0);

	/*
	 * Process should have INVALID_CONTEXT on all MMUs
	 */
	for (i = 0; i < max_mmu_ctxdoms; i++) {

		ASSERT(sfmmup->sfmmu_ctxs[i].cnum == INVALID_CONTEXT);
	}
#endif

	new_tsbinfo->tsb_next = old_tsbinfo->tsb_next;
	membar_stst();	/* strict ordering required */
	if (prevtsb)
		prevtsb->tsb_next = new_tsbinfo;
	else
		sfmmup->sfmmu_tsb = new_tsbinfo;
	membar_enter();	/* make sure new TSB globally visible */

	/*
	 * We need to migrate TSB entries from the old TSB to the new TSB
	 * if tsb_remap_ttes is set and the TSB is growing.
	 */
	if (tsb_remap_ttes && ((flags & TSB_GROW) == TSB_GROW))
		sfmmu_copy_tsb(old_tsbinfo, new_tsbinfo);

	SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);

	/*
	 * Drop the HAT lock to free our old tsb_info.
	 */
	sfmmu_hat_exit(hatlockp);

	if ((flags & TSB_GROW) == TSB_GROW) {
		SFMMU_STAT(sf_tsb_grow);
	} else if ((flags & TSB_SHRINK) == TSB_SHRINK) {
		SFMMU_STAT(sf_tsb_shrink);
```

```
        }

        sfmmu_tsbinfo_free(old_tsbinfo);

        (void) sfmmu_hat_enter(sfmmup);
        return (TSB_SUCCESS);
}
```
______***unchanged_portion_omitted_***

****
31077 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/os/syscall.c
6583 remove whole-process swapping
****
*______unchanged_portion_omitted_*

```
/*
 * Perform pre-system-call processing, including stopping for tracing,
 * auditing, microstate-accounting, etc.
 *
 * This routine is called only if the t_pre_sys flag is set.  Any condition
 * requiring pre-syscall handling must set the t_pre_sys flag.  If the
 * condition is persistent, this routine will repost t_pre_sys.
 */
int
pre_syscall(int arg0)
{
        unsigned int code;
        kthread_t *t = curthread;
        proc_t *p = ttoproc(t);
        klwp_t *lwp = ttolwp(t);
        struct regs *rp = lwptoregs(lwp);
        int     repost;

        t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

        ASSERT(t->t_schedflag & TS_DONT_SWAP);

        syscall_mstate(LMS_USER, LMS_SYSTEM);

        /*
         * The syscall arguments in the out registers should be pointed to
         * by lwp_ap.  If the args need to be copied so that the outs can
         * be changed without losing the ability to get the args for /proc,
         * they can be saved by save_syscall_args(), and lwp_ap will be
         * restored by post_syscall().
         */
        ASSERT(lwp->lwp_ap == (long *)&rp->r_o0);

        /*
         * Make sure the thread is holding the latest credentials for the
         * process.  The credentials in the process right now apply to this
         * thread for the entire system call.
         */
        if (t->t_cred != p->p_cred) {
                cred_t *oldcred = t->t_cred;
                /*
                 * DTrace accesses t_cred in probe context.  t_cred must
                 * always be either NULL, or point to a valid, allocated cred
                 * structure.
                 */
                t->t_cred = crgetcred();
                crfree(oldcred);
        }

        /*
         * Undo special arrangements to single-step the lwp
         * so that a debugger will see valid register contents.
         * Also so that the pc is valid for syncfpu().
         * Also so that a syscall like exec() can be stepped.
         */
        if (lwp->lwp_pcb.pcb_step != STEP_NONE) {
                (void) prundostep();
                repost = 1;
        }
```

```
        /*
         * Check for indirect system call in case we stop for tracing.
         * Don't allow multiple indirection.
         */
        code = t->t_sysnum;
        if (code == 0 && arg0 != 0) {           /* indirect syscall */
                code = arg0;
                t->t_sysnum = arg0;
        }

        /*
         * From the proc(4) manual page:
         * When entry to a system call is being traced, the traced process
         * stops after having begun the call to the system but before the
         * system call arguments have been fetched from the process.
         * If proc changes the args we must refetch them after starting.
         */
        if (PTOU(p)->u_systrap) {
                if (prismember(&PTOU(p)->u_entrymask, code)) {
                        /*
                         * Recheck stop condition, now that lock is held.
                         */
                        mutex_enter(&p->p_lock);
                        if (PTOU(p)->u_systrap &&
                            prismember(&PTOU(p)->u_entrymask, code)) {
                                stop(PR_SYSENTRY, code);
                                /*
                                 * Must refetch args since they were
                                 * possibly modified by /proc.  Indicate
                                 * that the valid copy is in the
                                 * registers.
                                 */
                                lwp->lwp_argsaved = 0;
                                lwp->lwp_ap = (long *)&rp->r_o0;
                        }
                        mutex_exit(&p->p_lock);
                }
                repost = 1;
        }

        if (lwp->lwp_sysabort) {
                /*
                 * lwp_sysabort may have been set via /proc while the process
                 * was stopped on PR_SYSENTRY.  If so, abort the system call.
                 * Override any error from the copyin() of the arguments.
                 */
                lwp->lwp_sysabort = 0;
                (void) set_errno(EINTR); /* sets post-sys processing */
                t->t_pre_sys = 1;       /* repost anyway */
                return (1);             /* don't do system call, return EINTR */
        }

        /* begin auditing for this syscall */
        if (audit_active == C2AUDIT_LOADED) {
                uint32_t auditing = au_zone_getstate(NULL);

                if (auditing & AU_AUDIT_MASK) {
                        int error;
                        if (error = audit_start(T_SYSCALL, code, auditing, \
                            0, lwp)) {
                                t->t_pre_sys = 1;       /* repost anyway */
                                lwp->lwp_error = 0;     /* for old drivers */
                                return (error);
                        }
                        repost = 1;
```

```
		}
	}

#ifndef NPROBE
	/* Kernel probe */
	if (tnf_tracing_active) {
		TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
			tnf_sysnum,	sysnum,		t->t_sysnum);
		t->t_post_sys = 1;	/* make sure post_syscall runs */
		repost = 1;
	}
#endif /* NPROBE */

#ifdef SYSCALLTRACE
	if (syscalltrace) {
		int i;
		long *ap;
		char *cp;
		char *sysname;
		struct sysent *callp;

		if (code >= NSYSCALL)
			callp = &nosys_ent;	/* nosys has no args */
		else
			callp = LWP_GETSYSENT(lwp) + code;
		(void) save_syscall_args();
		mutex_enter(&systrace_lock);
		printf("%d: ", p->p_pid);
		if (code >= NSYSCALL)
			printf("0x%x", code);
		else {
			sysname = mod_getsysname(code);
			printf("%s[0x%x]", sysname == NULL ? "NULL" :
			    sysname, code);
		}
		cp = "(";
		for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
			printf("%s%lx", cp, *ap);
			cp = ", ";
		}
		if (i)
			printf(")");
		printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
		mutex_exit(&systrace_lock);
	}
#endif /* SYSCALLTRACE */

	/*
	 * If there was a continuing reason for pre-syscall processing,
	 * set the t_pre_sys flag for the next system call.
	 */
	if (repost)
		t->t_pre_sys = 1;
	lwp->lwp_error = 0;	/* for old drivers */
	lwp->lwp_badpriv = PRIV_NONE;	/* for privilege tracing */
	return (0);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   50041 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/v9/os/v9dep.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

void
lwp_swapin(kthread_t *tp)
{
        struct machpcb *mpcb = lwptompcb(ttolwp(tp));

        mpcb->mpcb_pa = va_to_pa(mpcb);
        mpcb->mpcb_wbuf_pa = va_to_pa(mpcb->mpcb_wbuf);
}

/*
 * Construct the execution environment for the user's signal
 * handler and arrange for control to be given to it on return
 * to userland.  The library code now calls setcontext() to
 * clean up after the signal handler, so sigret() is no longer
 * needed.
 */
int
sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
{
        /*
         * 'volatile' is needed to ensure that values are
         * correct on the error return from on_fault().
         */
        volatile int minstacksz; /* min stack required to catch signal */
        int newstack = 0;       /* if true, switching to altstack */
        label_t ljb;
        caddr_t sp;
        struct regs *volatile rp;
        klwp_t *lwp = ttolwp(curthread);
        proc_t *volatile p = ttoproc(curthread);
        int fpq_size = 0;
        struct sigframe {
                struct frame frwin;
                ucontext_t uc;
        };
        siginfo_t *sip_addr;
        struct sigframe *volatile fp;
        ucontext_t *volatile tuc = NULL;
        char *volatile xregs = NULL;
        volatile size_t xregs_size = 0;
        gwindows_t *volatile gwp = NULL;
        volatile int gwin_size = 0;
        kfpu_t *fpp;
        struct machpcb *mpcb;
        volatile int watched = 0;
        volatile int watched2 = 0;
        caddr_t tos;

        /*
         * Make sure the current last user window has been flushed to
         * the stack save area before we change the sp.
         * Restore register window if a debugger modified it.
         */
        (void) flush_user_windows_to_stack(NULL);
        if (lwp->lwp_pcb.pcb_xregstat != XREGNONE)
                xregrestore(lwp, 0);

        mpcb = lwptompcb(lwp);
        rp = lwptoregs(lwp);
```

```
        /*
         * Clear the watchpoint return stack pointers.
         */
        mpcb->mpcb_rsp[0] = NULL;
        mpcb->mpcb_rsp[1] = NULL;

        minstacksz = sizeof (struct sigframe);

        /*
         * We know that sizeof (siginfo_t) is stack-aligned:
         * 128 bytes for ILP32, 256 bytes for LP64.
         */
        if (sip != NULL)
                minstacksz += sizeof (siginfo_t);

        /*
         * These two fields are pointed to by ABI structures and may
         * be of arbitrary length. Size them now so we know how big
         * the signal frame has to be.
         */
        fpp = lwptofpu(lwp);
        fpp->fpu_fprs = _fp_read_fprs();
        if ((fpp->fpu_en) || (fpp->fpu_fprs & FPRS_FEF)) {
                fpq_size = fpp->fpu_q_entrysize * fpp->fpu_qcnt;
                minstacksz += SA(fpq_size);
        }

        mpcb = lwptompcb(lwp);
        if (mpcb->mpcb_wbcnt != 0) {
                gwin_size = (mpcb->mpcb_wbcnt * sizeof (struct rwindow)) +
                    (SPARC_MAXREGWINDOW * sizeof (caddr_t)) + sizeof (long);
                minstacksz += SA(gwin_size);
        }

        /*
         * Extra registers, if support by this platform, may be of arbitrary
         * length. Size them now so we know how big the signal frame has to be.
         * For sparcv9 _LP64 user programs, use asrs instead of the xregs.
         */
        minstacksz += SA(xregs_size);

        /*
         * Figure out whether we will be handling this signal on
         * an alternate stack specified by the user. Then allocate
         * and validate the stack requirements for the signal handler
         * context. on_fault will catch any faults.
         */
        newstack = (sigismember(&PTOU(curproc)->u_signonstack, sig) &&
            !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE)));

        tos = (caddr_t)rp->r_sp + STACK_BIAS;
        /*
         * Force proper stack pointer alignment, even in the face of a
         * misaligned stack pointer from user-level before the signal.
         * Don't use the SA() macro because that rounds up, not down.
         */
        tos = (caddr_t)((uintptr_t)tos & ~(STACK_ALIGN - 1ul));

        if (newstack != 0) {
                fp = (struct sigframe *)
                    (SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
                    SA((int)lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN -
                    SA(minstacksz));
        } else {
                /*
```

```
		 * If we were unable to flush all register windows to
		 * the stack and we are not now on an alternate stack,
		 * just dump core with a SIGSEGV back in psig().
		 */
		if (sig == SIGSEGV &&
		    mpcb->mpcb_wbcnt != 0 &&
		    !(lwp->lwp_sigaltstack.ss_flags & SS_ONSTACK))
			return (0);
		fp = (struct sigframe *)(tos - SA(minstacksz));
		/*
		 * Could call grow here, but stack growth now handled below
		 * in code protected by on_fault().
		 */
	}
	sp = (caddr_t)fp + sizeof (struct sigframe);

	/*
	 * Make sure process hasn't trashed its stack.
	 */
	if ((caddr_t)fp >= p->p_usrstack ||
	    (caddr_t)fp + SA(minstacksz) >= p->p_usrstack) {
#ifdef DEBUG
		printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
		    PTOU(p)->u_comm, p->p_pid, sig);
		printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
		    (void *)fp, (void *)hdlr, rp->r_pc);
		printf("fp above USRSTACK\n");
#endif
		return (0);
	}

	watched = watch_disable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
	if (on_fault(&ljb))
		goto badstack;

	tuc = kmem_alloc(sizeof (ucontext_t), KM_SLEEP);
	savecontext(tuc, &lwp->lwp_sigoldmask);

	/*
	 * save extra register state if it exists
	 */
	if (xregs_size != 0) {
		xregs_setptr(lwp, tuc, sp);
		xregs = kmem_alloc(xregs_size, KM_SLEEP);
		xregs_get(lwp, xregs);
		copyout_noerr(xregs, sp, xregs_size);
		kmem_free(xregs, xregs_size);
		xregs = NULL;
		sp += SA(xregs_size);
	}

	copyout_noerr(tuc, &fp->uc, sizeof (*tuc));
	kmem_free(tuc, sizeof (*tuc));
	tuc = NULL;

	if (sip != NULL) {
		zoneid_t zoneid;

		uzero(sp, sizeof (siginfo_t));
		if (SI_FROMUSER(sip) &&
		    (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
		    zoneid != sip->si_zoneid) {
			k_siginfo_t sani_sip = *sip;
			sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
			sani_sip.si_uid = 0;
			sani_sip.si_ctid = -1;
```

```
			sani_sip.si_zoneid = zoneid;
			copyout_noerr(&sani_sip, sp, sizeof (sani_sip));
		} else {
			copyout_noerr(sip, sp, sizeof (*sip));
		}
		sip_addr = (siginfo_t *)sp;
		sp += sizeof (siginfo_t);

		if (sig == SIGPROF &&
		    curthread->t_rprof != NULL &&
		    curthread->t_rprof->rp_anystate) {
			/*
			 * We stand on our head to deal with
			 * the real time profiling signal.
			 * Fill in the stuff that doesn't fit
			 * in a normal k_siginfo structure.
			 */
			int i = sip->si_nsysarg;
			while (--i >= 0) {
				sulword_noerr(
				    (ulong_t *)&sip_addr->si_sysarg[i],
				    (ulong_t)lwp->lwp_arg[i]);
			}
			copyout_noerr(curthread->t_rprof->rp_state,
			    sip_addr->si_mstate,
			    sizeof (curthread->t_rprof->rp_state));
		}
	} else {
		sip_addr = (siginfo_t *)NULL;
	}

	/*
	 * When flush_user_windows_to_stack() can't save all the
	 * windows to the stack, it puts them in the lwp's pcb.
	 */
	if (gwin_size != 0) {
		gwp = kmem_alloc(gwin_size, KM_SLEEP);
		getgwins(lwp, gwp);
		sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)sp);
		copyout_noerr(gwp, sp, gwin_size);
		kmem_free(gwp, gwin_size);
		gwp = NULL;
		sp += SA(gwin_size);
	} else
		sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)NULL);

	if (fpq_size != 0) {
		struct _fq *fqp = (struct _fq *)sp;
		sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)fqp);
		copyout_noerr(mpcb->mpcb_fpu_q, fqp, fpq_size);

		/*
		 * forget the fp queue so that the signal handler can run
		 * without being harrassed--it will do a setcontext that will
		 * re-establish the queue if there still is one
		 *
		 * NOTE: fp_runq() relies on the qcnt field being zeroed here
		 *      to terminate its processing of the queue after signal
		 *      delivery.
		 */
		mpcb->mpcb_fpu->fpu_qcnt = 0;
		sp += SA(fpq_size);

		/* Also, syscall needs to know about this */
		mpcb->mpcb_flags |= FP_TRAPPED;
```

```
        } else {
                sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)NULL);
                suword8_noerr(&fp->uc.uc_mcontext.fpregs.fpu_qcnt, 0);
        }


        /*
         * Since we flushed the user's windows and we are changing his
         * stack pointer, the window that the user will return to will
         * be restored from the save area in the frame we are setting up.
         * We copy in save area for old stack pointer so that debuggers
         * can do a proper stack backtrace from the signal handler.
         */
        if (mpcb->mpcb_wbcnt == 0) {
                watched2 = watch_disable_addr(tos, sizeof (struct rwindow),
                    S_READ);
                ucopy(tos, &fp->frwin, sizeof (struct rwindow));
        }

        lwp->lwp_oldcontext = (uintptr_t)&fp->uc;

        if (newstack != 0) {
                lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;

                if (lwp->lwp_ustack) {
                        copyout_noerr(&lwp->lwp_sigaltstack,
                            (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
                }
        }

        no_fault();
        mpcb->mpcb_wbcnt = 0;           /* let user go on */

        if (watched2)
                watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
        if (watched)
                watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);

        /*
         * Set up user registers for execution of signal handler.
         */
        rp->r_sp = (uintptr_t)fp - STACK_BIAS;
        rp->r_pc = (uintptr_t)hdlr;
        rp->r_npc = (uintptr_t)hdlr + 4;
        /* make sure %asi is ASI_PNF */
        rp->r_tstate &= ~((uint64_t)TSTATE_ASI_MASK << TSTATE_ASI_SHIFT);
        rp->r_tstate |= ((uint64_t)ASI_PNF << TSTATE_ASI_SHIFT);
        rp->r_o0 = sig;
        rp->r_o1 = (uintptr_t)sip_addr;
        rp->r_o2 = (uintptr_t)&fp->uc;
        /*
         * Don't set lwp_eosys here.  sendsig() is called via psig() after
         * lwp_eosys is handled, so setting it here would affect the next
         * system call.
         */
        return (1);

badstack:
        no_fault();
        if (watched2)
                watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
        if (watched)
                watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
        if (tuc)
                kmem_free(tuc, sizeof (ucontext_t));
        if (xregs)
```

```
                kmem_free(xregs, xregs_size);
        if (gwp)
                kmem_free(gwp, gwin_size);
#ifdef DEBUG
        printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
            PTOU(p)->u_comm, p->p_pid, sig);
        printf("on fault, sigsp = %p, action = %p, upc = 0x%lx\n",
            (void *)fp, (void *)hdlr, rp->r_pc);
#endif
        return (0);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   12166 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/v9/vm/seg_nf.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T */
/* All Rights Reserved */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * VM - segment for non-faulting loads.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/mman.h>
#include <sys/errno.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vnode.h>
#include <sys/proc.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/archsystm.h>
#include <sys/lgrp.h>

#include <vm/page.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/vpage.h>

/*
 * Private seg op routines.
 */
static int      segnf_dup(struct seg *seg, struct seg *newseg);
```

```
static int      segnf_unmap(struct seg *seg, caddr_t addr, size_t len);
static void     segnf_free(struct seg *seg);
static faultcode_t segnf_nomap(void);
static int      segnf_setprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static int      segnf_checkprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static void     segnf_badop(void);
static int      segnf_nop(void);
static int      segnf_getprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t *protv);
static u_offset_t segnf_getoffset(struct seg *seg, caddr_t addr);
static int      segnf_gettype(struct seg *seg, caddr_t addr);
static int      segnf_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
static void     segnf_dump(struct seg *seg);
static int      segnf_pagelock(struct seg *seg, caddr_t addr, size_t len,
                    struct page ***ppp, enum lock_type type, enum seg_rw rw);
static int      segnf_setpagesize(struct seg *seg, caddr_t addr, size_t len,
                    uint_t szc);
static int      segnf_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
static lgrp_mem_policy_info_t   *segnf_getpolicy(struct seg *seg,
    caddr_t addr);


struct seg_ops segnf_ops = {
        segnf_dup,
        segnf_unmap,
        segnf_free,
        (faultcode_t (*)(struct hat *, struct seg *, caddr_t, size_t,
            enum fault_type, enum seg_rw))
                segnf_nomap,            /* fault */
        (faultcode_t (*)(struct seg *, caddr_t))
                segnf_nomap,            /* faulta */
        segnf_setprot,
        segnf_checkprot,
        (int (*)())segnf_badop,         /* kluster */
        (size_t (*)(struct seg *))NULL, /* swapout */
        (int (*)(struct seg *, caddr_t, size_t, int, uint_t))
                segnf_nop,              /* sync */
        (size_t (*)(struct seg *, caddr_t, size_t, char *))
                segnf_nop,              /* incore */
        (int (*)(struct seg *, caddr_t, size_t, int, int, ulong_t *, size_t))
                segnf_nop,              /* lockop */
        segnf_getprot,
        segnf_getoffset,
        segnf_gettype,
        segnf_getvp,
        (int (*)(struct seg *, caddr_t, size_t, uint_t))
                segnf_nop,              /* advise */
        segnf_dump,
        segnf_pagelock,
        segnf_setpagesize,
        segnf_getmemid,
        segnf_getpolicy,
};
______unchanged_portion_omitted_
```

```
**********************************************************
   14116 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sun4/os/mlsetup.c
6583 remove whole-process swapping
**********************************************************
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/disp.h>
#include <sys/autoconf.h>
#include <sys/promif.h>
#include <sys/prom_plat.h>
#include <sys/promimpl.h>
#include <sys/platform_module.h>
#include <sys/clock.h>
#include <sys/pte.h>
#include <sys/scb.h>
#include <sys/cpu.h>
#include <sys/stack.h>
#include <sys/intreg.h>
#include <sys/ivintr.h>
#include <vm/as.h>
#include <vm/hat_sfmmu.h>
#include <sys/reboot.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/trap.h>
#include <sys/machtrap.h>
#include <sys/privregs.h>
#include <sys/machpcb.h>
#include <sys/proc.h>
#include <sys/cpupart.h>
#include <sys/pset.h>
#include <sys/cpu_module.h>
#include <sys/copyops.h>
#include <sys/panic.h>
#include <sys/bootconf.h>       /* for bootops */
#include <sys/pg.h>
#include <sys/kdi.h>
#include <sys/fpras.h>
```

```
#include <sys/prom_debug.h>
#include <sys/debug.h>

#include <sys/sunddi.h>
#include <sys/lgrp.h>
#include <sys/traptrace.h>

#include <sys/kobj_impl.h>
#include <sys/kdi_machimpl.h>

/*
 * External Routines:
 */
extern void map_wellknown_devices(void);
extern void hsvc_setup(void);
extern void mach_descrip_startup_init(void);
extern void mach_soft_state_init(void);

int     dcache_size;
int     dcache_linesize;
int     icache_size;
int     icache_linesize;
int     ecache_size;
int     ecache_alignsize;
int     ecache_associativity;
int     ecache_setsize;                 /* max possible e$ setsize */
int     cpu_setsize;                    /* max e$ setsize of configured cpus */
int     dcache_line_mask;               /* spitfire only */
int     vac_size;                       /* cache size in bytes */
uint_t  vac_mask;                       /* VAC alignment consistency mask */
int     vac_shift;                      /* log2(vac_size) for ppmapout() */
int     vac = 0;        /* virtual address cache type (none == 0) */

/*
 * fpRAS.  An individual sun4* machine class (or perhaps subclass,
 * eg sun4u/cheetah) must set fpras_implemented to indicate that it implements
 * the fpRAS feature.  The feature can be suppressed by setting fpras_disable
 * or the mechanism can be disabled for individual copy operations with
 * fpras_disableids.  All these are checked in post_startup() code so
 * fpras_disable and fpras_disableids can be set in /etc/system.
 * If/when fpRAS is implemented on non-sun4 architectures these
 * definitions will need to move up to the common level.
 */
int     fpras_implemented;
int     fpras_disable;
int     fpras_disableids;

/*
 * Static Routines:
 */
static void kern_splr_preprom(void);
static void kern_splx_postprom(void);

/*
 * Setup routine called right before main(). Interposing this function
 * before main() allows us to call it in a machine-independent fashion.
 */

void
mlsetup(struct regs *rp, kfpu_t *fp)
{
        struct machpcb *mpcb;

        extern char t0stack[];
        extern struct classfuncs sys_classfuncs;
        extern disp_t cpu0_disp;
        unsigned long long pa;

#ifdef TRAPTRACE
        TRAP_TRACE_CTL *ctlp;
#endif /* TRAPTRACE */

        /* drop into kmdb on boot -d */
        if (boothowto & RB_DEBUGENTER)
                kmdb_enter();

        /*
         * initialize cpu_self
         */
        cpu0.cpu_self = &cpu0;

        /*
         * initialize t0
         */
        t0.t_stk = (caddr_t)rp - REGOFF;
        /* Can't use va_to_pa here - wait until prom_ initialized */
        t0.t_stkbase = t0stack;
        t0.t_pri = maxclsyspri - 3;
        t0.t_schedflag = 0;
        t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
        t0.t_procp = &p0;
        t0.t_plockp = &p0lock.pl_lock;
        t0.t_lwp = &lwp0;
        t0.t_forw = &t0;
        t0.t_back = &t0;
        t0.t_next = &t0;
        t0.t_prev = &t0;
        t0.t_cpu = &cpu0;                       /* loaded by _start */
        t0.t_disp_queue = &cpu0_disp;
        t0.t_bind_cpu = PBIND_NONE;
        t0.t_bind_pset = PS_NONE;
        t0.t_bindflag = (uchar_t)default_binding_mode;
        t0.t_cpupart = &cp_default;
        t0.t_clfuncs = &sys_classfuncs.thread;
        t0.t_copyops = NULL;
        THREAD_ONPROC(&t0, CPU);

        lwp0.lwp_thread = &t0;
        lwp0.lwp_procp = &p0;
        lwp0.lwp_regs = (void *)rp;
        t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

        mpcb = lwptompcb(&lwp0);
        mpcb->mpcb_fpu = fp;
        mpcb->mpcb_fpu->fpu_q = mpcb->mpcb_fpu_q;
        mpcb->mpcb_thread = &t0;
        lwp0.lwp_fpu = (void *)mpcb->mpcb_fpu;

        p0.p_exec = NULL;
        p0.p_stat = SRUN;
        p0.p_flag = SSYS;
        p0.p_tlist = &t0;
        p0.p_stksize = 2*PAGESIZE;
        p0.p_stkpageszc = 0;
        p0.p_as = &kas;
        p0.p_lockp = &p0lock;
        p0.p_utraps = NULL;
        p0.p_brkpageszc = 0;
        p0.p_t1_lgrpid = LGRP_NONE;
        p0.p_tr_lgrpid = LGRP_NONE;
        sigorset(&p0.p_ignore, &ignoredefault);
```

```

        CPU->cpu_thread = &t0;
        CPU->cpu_dispthread = &t0;
        bzero(&cpu0_disp, sizeof (disp_t));
        CPU->cpu_disp = &cpu0_disp;
        CPU->cpu_disp->disp_cpu = CPU;
        CPU->cpu_idle_thread = &t0;
        CPU->cpu_flags = CPU_RUNNING;
        CPU->cpu_id = getprocessorid();
        CPU->cpu_dispatch_pri = t0.t_pri;

        /*
         * Initialize thread/cpu microstate accounting
         */
        init_mstate(&t0, LMS_SYSTEM);
        init_cpu_mstate(CPU, CMS_SYSTEM);

        /*
         * Initialize lists of available and active CPUs.
         */
        cpu_list_init(CPU);

        cpu_vm_data_init(CPU);

        pg_cpu_bootstrap(CPU);

        (void) prom_set_preprom(kern_splr_preprom);
        (void) prom_set_postprom(kern_splx_postprom);
        PRM_INFO("mlsetup: now ok to call prom_printf");

        mpcb->mpcb_pa = va_to_pa(t0.t_stk);

        /*
         * Claim the physical and virtual resources used by panicbuf,
         * then map panicbuf.  This operation removes the phys and
         * virtual addresses from the free lists.
         */
        if (prom_claim_virt(PANICBUFSIZE, panicbuf) != panicbuf)
                prom_panic("Can't claim panicbuf virtual address");

        if (prom_retain("panicbuf", PANICBUFSIZE, MMU_PAGESIZE, &pa) != 0)
                prom_panic("Can't allocate retained panicbuf physical address");

        if (prom_map_phys(-1, PANICBUFSIZE, panicbuf, pa) != 0)
                prom_panic("Can't map panicbuf");

        PRM_DEBUG(panicbuf);
        PRM_DEBUG(pa);

        /*
         * Negotiate hypervisor services, if any
         */
        hsvc_setup();
        mach_soft_state_init();

#ifdef TRAPTRACE
        /*
         * initialize the trap trace buffer for the boot cpu
         * XXX todo, dynamically allocate this buffer too
         */
        ctlp = &trap_trace_ctl[CPU->cpu_id];
        ctlp->d.vaddr_base = trap_tr0;
        ctlp->d.offset = ctlp->d.last_offset = 0;
        ctlp->d.limit = TRAP_TSIZE;             /* XXX dynamic someday */
        ctlp->d.paddr_base = va_to_pa(trap_tr0);
#endif /* TRAPTRACE */
```

```
	/*
	 * Initialize the Machine Description kernel framework
	 */

	mach_descrip_startup_init();

	/*
	 * initialize HV trap trace buffer for the boot cpu
	 */
	mach_htraptrace_setup(CPU->cpu_id);
	mach_htraptrace_configure(CPU->cpu_id);

	/*
	 * lgroup framework initialization. This must be done prior
	 * to devices being mapped.
	 */
	lgrp_init(LGRP_INIT_STAGE1);

	cpu_setup();

	if (boothowto & RB_HALT) {
		prom_printf("unix: kernel halted by -h flag\n");
		prom_enter_mon();
	}

	setcputype();
	map_wellknown_devices();
	setcpudelay();
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   51352 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sun4/os/trap.c
6583 remove whole-process swapping
**********************************************************
______unchanged_portion_omitted_

#if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
int     ill_calls;
#endif

/*
 * Currently, the only PREFETCH/PREFETCHA instructions which cause traps
 * are the "strong" prefetches (fcn=20-23).  But we check for all flavors of
 * PREFETCH, in case some future variant also causes a DATA_MMU_MISS.
 */
#define IS_PREFETCH(i)  (((i) & 0xc1780000) == 0xc1680000)

#define IS_FLUSH(i)     (((i) & 0xc1f80000) == 0x81d80000)
#define IS_SWAP(i)      (((i) & 0xc1f80000) == 0xc0780000)
#define IS_LDSTUB(i)    (((i) & 0xc1f80000) == 0xc0680000)
#define IS_FLOAT(i)     (((i) & 0x1000000) != 0)
#define IS_STORE(i)     (((i) >> 21) & 1)

/*
 * Called from the trap handler when a processor trap occurs.
 */
/*VARARGS2*/
void
trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t mmu_fsr)
{
        proc_t *p = ttoproc(curthread);
        klwp_id_t lwp = ttolwp(curthread);
        struct machpcb *mpcb = NULL;
        k_siginfo_t siginfo;
        uint_t op3, fault = 0;
        int stepped = 0;
        greg_t oldpc;
        int mstate;
        char *badaddr;
        faultcode_t res;
        enum fault_type fault_type;
        enum seg_rw rw;
        uintptr_t lofault;
        label_t *onfault;
        int instr;
        int iskernel;
        int watchcode;
        int watchpage;
        extern faultcode_t pagefault(caddr_t, enum fault_type,
            enum seg_rw, int);
#ifdef sun4v
        extern boolean_t tick_stick_emulation_active;
#endif  /* sun4v */

        CPU_STATS_ADDQ(CPU, sys, trap, 1);

#ifdef SF_ERRATA_23 /* call causes illegal-insn */
        ASSERT((curthread->t_schedflag & TS_DONT_SWAP) ||
            (type == T_UNIMP_INSTR));
#else
        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
#endif /* SF_ERRATA_23 */

        if (USERMODE(rp->r_tstate) || (type & T_USER)) {
                /*
                 * Set lwp_state before trying to acquire any
                 * adaptive lock
                 */
                ASSERT(lwp != NULL);
                lwp->lwp_state = LWP_SYS;
                /*
                 * Set up the current cred to use during this trap. u_cred
                 * no longer exists.  t_cred is used instead.
                 * The current process credential applies to the thread for
                 * the entire trap.  If trapping from the kernel, this
                 * should already be set up.
                 */
                if (curthread->t_cred != p->p_cred) {
                        cred_t *oldcred = curthread->t_cred;
                        /*
                         * DTrace accesses t_cred in probe context.  t_cred
                         * must always be either NULL, or point to a valid,
                         * allocated cred structure.
                         */
                        curthread->t_cred = crgetcred();
                        crfree(oldcred);
                }
                type |= T_USER;
                ASSERT((type == (T_SYS_RTT_PAGE | T_USER)) ||
                    (type == (T_SYS_RTT_ALIGN | T_USER)) ||
                    lwp->lwp_regs == rp);
                mpcb = lwptompcb(lwp);
                switch (type) {
                case T_WIN_OVERFLOW + T_USER:
                case T_WIN_UNDERFLOW + T_USER:
                case T_SYS_RTT_PAGE + T_USER:
                case T_DATA_MMU_MISS + T_USER:
                        mstate = LMS_DFAULT;
                        break;
                case T_INSTR_MMU_MISS + T_USER:
                        mstate = LMS_TFAULT;
                        break;
                default:
                        mstate = LMS_TRAP;
                        break;
                }
                /* Kernel probe */
                TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
                    tnf_microstate, state, (char)mstate);
                mstate = new_mstate(curthread, mstate);
                siginfo.si_signo = 0;
                stepped =
                    lwp->lwp_pcb.pcb_step != STEP_NONE &&
                    ((oldpc = rp->r_pc), prundostep()) &&
                    mmu_btop((uintptr_t)addr) == mmu_btop((uintptr_t)oldpc);
                /* this assignment must not precede call to prundostep() */
                oldpc = rp->r_pc;
        }

        TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
            "C_trap_handler_enter:type %x", type);

#ifdef  F_DEFERRED
        /*
         * Take any pending floating point exceptions now.
         * If the floating point unit has an exception to handle,
         * just return to user-level to let the signal handler run.
         * The instruction that got us to trap() will be reexecuted on
         * return from the signal handler and we will trap to here again.
         * This is necessary to disambiguate simultaneous traps which
         * happen when a floating-point exception is pending and a
```

```
	 * machine fault is incurred.
	 */
	if (type & USER) {
		/*
		 * FP_TRAPPED is set only by sendsig() when it copies
		 * out the floating-point queue for the signal handler.
		 * It is set there so we can test it here and in syscall().
		 */
		mpcb->mpcb_flags &= ~FP_TRAPPED;
		syncfpu();
		if (mpcb->mpcb_flags & FP_TRAPPED) {
			/*
			 * trap() has have been called recursively and may
			 * have stopped the process, so do single step
			 * support for /proc.
			 */
			mpcb->mpcb_flags &= ~FP_TRAPPED;
			goto out;
		}
	}
#endif
	switch (type) {
		case T_DATA_MMU_MISS:
		case T_INSTR_MMU_MISS + T_USER:
		case T_DATA_MMU_MISS + T_USER:
		case T_DATA_PROT + T_USER:
		case T_AST + T_USER:
		case T_SYS_RTT_PAGE + T_USER:
		case T_FLUSH_PCB + T_USER:
		case T_FLUSHW + T_USER:
			break;

		default:
			FTRACE_3("trap(): type=0x%lx, regs=0x%lx, addr=0x%lx",
			    (ulong_t)type, (ulong_t)rp, (ulong_t)addr);
			break;
	}

	switch (type) {

	default:
		/*
		 * Check for user software trap.
		 */
		if (type & T_USER) {
			if (tudebug)
				showregs(type, rp, (caddr_t)0, 0);
			if ((type & ~T_USER) >= T_SOFTWARE_TRAP) {
				bzero(&siginfo, sizeof (siginfo));
				siginfo.si_signo = SIGILL;
				siginfo.si_code  = ILL_ILLTRP;
				siginfo.si_addr  = (caddr_t)rp->r_pc;
				siginfo.si_trapno = type &~ T_USER;
				fault = FLTILL;
				break;
			}
		}
		addr = (caddr_t)rp->r_pc;
		(void) die(type, rp, addr, 0);
		/*NOTREACHED*/

	case T_ALIGNMENT:	/* supv alignment error */
		if (nfload(rp, NULL))
			goto cleanup;

		if (curthread->t_lofault) {
```

```
			if (lodebug) {
				showregs(type, rp, addr, 0);
				traceback((caddr_t)rp->r_sp);
			}
			rp->r_g1 = EFAULT;
			rp->r_pc = curthread->t_lofault;
			rp->r_npc = rp->r_pc + 4;
			goto cleanup;
		}
		(void) die(type, rp, addr, 0);
		/*NOTREACHED*/

	case T_INSTR_EXCEPTION:		/* sys instruction access exception */
		addr = (caddr_t)rp->r_pc;
		(void) die(type, rp, addr, mmu_fsr);
		/*NOTREACHED*/

	case T_INSTR_MMU_MISS:		/* sys instruction mmu miss */
		addr = (caddr_t)rp->r_pc;
		(void) die(type, rp, addr, 0);
		/*NOTREACHED*/

	case T_DATA_EXCEPTION:		/* system data access exception */
		switch (X_FAULT_TYPE(mmu_fsr)) {
		case FT_RANGE:
			/*
			 * This happens when we attempt to dereference an
			 * address in the address hole.  If t_ontrap is set,
			 * then break and fall through to T_DATA_MMU_MISS /
			 * T_DATA_PROT case below.  If lofault is set, then
			 * honour it (perhaps the user gave us a bogus
			 * address in the hole to copyin from or copyout to?)
			 */

			if (curthread->t_ontrap != NULL)
				break;

			addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
			if (curthread->t_lofault) {
				if (lodebug) {
					showregs(type, rp, addr, 0);
					traceback((caddr_t)rp->r_sp);
				}
				rp->r_g1 = EFAULT;
				rp->r_pc = curthread->t_lofault;
				rp->r_npc = rp->r_pc + 4;
				goto cleanup;
			}
			(void) die(type, rp, addr, mmu_fsr);
			/*NOTREACHED*/

		case FT_PRIV:
			/*
			 * This can happen if we access ASI_USER from a kernel
			 * thread.  To support pxfs, we need to honor lofault if
			 * we're doing a copyin/copyout from a kernel thread.
			 */

			if (nfload(rp, NULL))
				goto cleanup;
			addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
			if (curthread->t_lofault) {
				if (lodebug) {
					showregs(type, rp, addr, 0);
					traceback((caddr_t)rp->r_sp);
				}
```

```
                                rp->r_g1 = EFAULT;
                                rp->r_pc = curthread->t_lofault;
                                rp->r_npc = rp->r_pc + 4;
                                goto cleanup;
                        }
                        (void) die(type, rp, addr, mmu_fsr);
                        /*NOTREACHED*/

                default:
                        if (nfload(rp, NULL))
                                goto cleanup;
                        addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
                        (void) die(type, rp, addr, mmu_fsr);
                        /*NOTREACHED*/

                case FT_NFO:
                        break;
                }
                /* fall into ... */

        case T_DATA_MMU_MISS:           /* system data mmu miss */
        case T_DATA_PROT:               /* system data protection fault */
                if (nfload(rp, &instr))
                        goto cleanup;

                /*
                 * If we're under on_trap() protection (see <sys/ontrap.h>),
                 * set ot_trap and return from the trap to the trampoline.
                 */
                if (curthread->t_ontrap != NULL) {
                        on_trap_data_t *otp = curthread->t_ontrap;

                        TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT,
                            "C_trap_handler_exit");
                        TRACE_0(TR_FAC_TRAP, TR_TRAP_END, "trap_end");

                        if (otp->ot_prot & OT_DATA_ACCESS) {
                                otp->ot_trap |= OT_DATA_ACCESS;
                                rp->r_pc = otp->ot_trampoline;
                                rp->r_npc = rp->r_pc + 4;
                                goto cleanup;
                        }
                }
                lofault = curthread->t_lofault;
                onfault = curthread->t_onfault;
                curthread->t_lofault = 0;

                mstate = new_mstate(curthread, LMS_KFAULT);

                switch (type) {
                case T_DATA_PROT:
                        fault_type = F_PROT;
                        rw = S_WRITE;
                        break;
                case T_INSTR_MMU_MISS:
                        fault_type = F_INVAL;
                        rw = S_EXEC;
                        break;
                case T_DATA_MMU_MISS:
                case T_DATA_EXCEPTION:
                        /*
                         * The hardware doesn't update the sfsr on mmu
                         * misses so it is not easy to find out whether
                         * the access was a read or a write so we need
                         * to decode the actual instruction.
                         */
```

```
                        fault_type = F_INVAL;
                        rw = get_accesstype(rp);
                        break;
                default:
                        cmn_err(CE_PANIC, "trap: unknown type %x", type);
                        break;
                }
                /*
                 * We determine if access was done to kernel or user
                 * address space.  The addr passed into trap is really the
                 * tag access register.
                 */
                iskernel = (((uintptr_t)addr & TAGACC_CTX_MASK) == KCONTEXT);
                addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);

                res = pagefault(addr, fault_type, rw, iskernel);
                if (!iskernel && res == FC_NOMAP &&
                    addr < p->p_usrstack && grow(addr))
                        res = 0;

                (void) new_mstate(curthread, mstate);

                /*
                 * Restore lofault and onfault.  If we resolved the fault, exit.
                 * If we didn't and lofault wasn't set, die.
                 */
                curthread->t_lofault = lofault;
                curthread->t_onfault = onfault;

                if (res == 0)
                        goto cleanup;

                if (IS_PREFETCH(instr)) {
                        /* skip prefetch instructions in kernel-land */
                        rp->r_pc = rp->r_npc;
                        rp->r_npc += 4;
                        goto cleanup;
                }

                if ((lofault == 0 || lodebug) &&
                    (calc_memaddr(rp, &badaddr) == SIMU_SUCCESS))
                        addr = badaddr;
                if (lofault == 0)
                        (void) die(type, rp, addr, 0);
                /*
                 * Cannot resolve fault.  Return to lofault.
                 */
                if (lodebug) {
                        showregs(type, rp, addr, 0);
                        traceback((caddr_t)rp->r_sp);
                }
                if (FC_CODE(res) == FC_OBJERR)
                        res = FC_ERRNO(res);
                else
                        res = EFAULT;
                rp->r_g1 = res;
                rp->r_pc = curthread->t_lofault;
                rp->r_npc = curthread->t_lofault + 4;
                goto cleanup;

        case T_INSTR_EXCEPTION + T_USER: /* user insn access exception */
                bzero(&siginfo, sizeof (siginfo));
                siginfo.si_addr = (caddr_t)rp->r_pc;
                siginfo.si_signo = SIGSEGV;
                siginfo.si_code = X_FAULT_TYPE(mmu_fsr) == FT_PRIV ?
                    SEGV_ACCERR : SEGV_MAPERR;
                fault = FLTBOUNDS;
                break;

        case T_WIN_OVERFLOW + T_USER:   /* window overflow in ??? */
        case T_WIN_UNDERFLOW + T_USER:  /* window underflow in ??? */
        case T_SYS_RTT_PAGE + T_USER:   /* window underflow in user_rtt */
        case T_INSTR_MMU_MISS + T_USER: /* user instruction mmu miss */
        case T_DATA_MMU_MISS + T_USER:  /* user data mmu miss */
        case T_DATA_PROT + T_USER:      /* user data protection fault */
                switch (type) {
                case T_INSTR_MMU_MISS + T_USER:
                        addr = (caddr_t)rp->r_pc;
                        fault_type = F_INVAL;
                        rw = S_EXEC;
                        break;

                case T_DATA_MMU_MISS + T_USER:
                        addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
                        fault_type = F_INVAL;
                        /*
                         * The hardware doesn't update the sfsr on mmu misses
                         * so it is not easy to find out whether the access
                         * was a read or a write so we need to decode the
                         * actual instruction.  XXX BUGLY HW
                         */
                        rw = get_accesstype(rp);
                        break;

                case T_DATA_PROT + T_USER:
                        addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
                        fault_type = F_PROT;
                        rw = S_WRITE;
                        break;

                case T_WIN_OVERFLOW + T_USER:
                        addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
                        fault_type = F_INVAL;
                        rw = S_WRITE;
                        break;

                case T_WIN_UNDERFLOW + T_USER:
                case T_SYS_RTT_PAGE + T_USER:
                        addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
                        fault_type = F_INVAL;
                        rw = S_READ;
                        break;

                default:
                        cmn_err(CE_PANIC, "trap: unknown type %x", type);
                        break;
                }

                /*
                 * If we are single stepping do not call pagefault
                 */
                if (stepped) {
                        res = FC_NOMAP;
                } else {
                        caddr_t vaddr = addr;
                        size_t sz;
                        int ta;

                        ASSERT(!(curthread->t_flag & T_WATCHPT));
                        watchpage = (pr_watch_active(p) &&
                            type != T_WIN_OVERFLOW + T_USER &&
                            type != T_WIN_UNDERFLOW + T_USER &&
```

```
                            type != T_SYS_RTT_PAGE + T_USER &&
                            pr_is_watchpage(addr, rw));

                        if (!watchpage ||
                            (sz = instr_size(rp, &vaddr, rw)) <= 0)
                                /* EMPTY */;
                        else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
                            sz, NULL, rw)) != 0) {
                                if (ta) {
                                        do_watch_step(vaddr, sz, rw,
                                            watchcode, rp->r_pc);
                                        fault_type = F_INVAL;
                                } else {
                                        bzero(&siginfo, sizeof (siginfo));
                                        siginfo.si_signo = SIGTRAP;
                                        siginfo.si_code = watchcode;
                                        siginfo.si_addr = vaddr;
                                        siginfo.si_trapafter = 0;
                                        siginfo.si_pc = (caddr_t)rp->r_pc;
                                        fault = FLTWATCH;
                                        break;
                                }
                        } else {
                                if (rw != S_EXEC &&
                                    pr_watch_emul(rp, vaddr, rw))
                                        goto out;
                                do_watch_step(vaddr, sz, rw, 0, 0);
                                fault_type = F_INVAL;
                        }

                        if (pr_watch_active(p) &&
                            (type == T_WIN_OVERFLOW + T_USER ||
                            type == T_WIN_UNDERFLOW + T_USER ||
                            type == T_SYS_RTT_PAGE + T_USER)) {
                                int dotwo = (type == T_WIN_UNDERFLOW + T_USER);
                                if (copy_return_window(dotwo))
                                        goto out;
                                fault_type = F_INVAL;
                        }

                        res = pagefault(addr, fault_type, rw, 0);

                        /*
                         * If pagefault succeed, ok.
                         * Otherwise grow the stack automatically.
                         */
                        if (res == 0 ||
                            (res == FC_NOMAP &&
                            type != T_INSTR_MMU_MISS + T_USER &&
                            addr < p->p_usrstack &&
                            grow(addr))) {
                                int ismem = prismember(&p->p_fltmask, FLTPAGE);

                                /*
                                 * instr_size() is used to get the exact
                                 * address of the fault, instead of the
                                 * page of the fault. Unfortunately it is
                                 * very slow, and this is an important
                                 * code path. Don't call it unless
                                 * correctness is needed. ie. if FLTPAGE
                                 * is set, or we're profiling.
                                 */

                                if (curthread->t_rprof != NULL || ismem)
                                        (void) instr_size(rp, &addr, rw);
```

```
				lwp->lwp_lastfault = FLTPAGE;
				lwp->lwp_lastfaddr = addr;

				if (ismem) {
					bzero(&siginfo, sizeof (siginfo));
					siginfo.si_addr = addr;
					(void) stop_on_fault(FLTPAGE, &siginfo);
				}
				goto out;
			}

			if (type != (T_INSTR_MMU_MISS + T_USER)) {
				/*
				 * check for non-faulting loads, also
				 * fetch the instruction to check for
				 * flush
				 */
				if (nfload(rp, &instr))
					goto out;

				/* skip userland prefetch instructions */
				if (IS_PREFETCH(instr)) {
					rp->r_pc = rp->r_npc;
					rp->r_npc += 4;
					goto out;
					/*NOTREACHED*/
				}

				/*
				 * check if the instruction was a
				 * flush.  ABI allows users to specify
				 * an illegal address on the flush
				 * instruction so we simply return in
				 * this case.
				 *
				 * NB: the hardware should set a bit
				 * indicating this trap was caused by
				 * a flush instruction.  Instruction
				 * decoding is bugly!
				 */
				if (IS_FLUSH(instr)) {
					/* skip the flush instruction */
					rp->r_pc = rp->r_npc;
					rp->r_npc += 4;
					goto out;
					/*NOTREACHED*/
				}
			} else if (res == FC_PROT) {
				report_stack_exec(p, addr);
			}

			if (tudebug)
				showregs(type, rp, addr, 0);
		}

		/*
		 * In the case where both pagefault and grow fail,
		 * set the code to the value provided by pagefault.
		 */
		(void) instr_size(rp, &addr, rw);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_addr = addr;
		if (FC_CODE(res) == FC_OBJERR) {
			siginfo.si_errno = FC_ERRNO(res);
			if (siginfo.si_errno != EINTR) {
				siginfo.si_signo = SIGBUS;
```

```
				siginfo.si_code = BUS_OBJERR;
				fault = FLTACCESS;
			}
		} else { /* FC_NOMAP || FC_PROT */
			siginfo.si_signo = SIGSEGV;
			siginfo.si_code = (res == FC_NOMAP) ?
			    SEGV_MAPERR : SEGV_ACCERR;
			fault = FLTBOUNDS;
		}
		/*
		 * If this is the culmination of a single-step,
		 * reset the addr, code, signal and fault to
		 * indicate a hardware trace trap.
		 */
		if (stepped) {
			pcb_t *pcb = &lwp->lwp_pcb;

			siginfo.si_signo = 0;
			fault = 0;
			if (pcb->pcb_step == STEP_WASACTIVE) {
				pcb->pcb_step = STEP_NONE;
				pcb->pcb_tracepc = NULL;
				oldpc = rp->r_pc - 4;
			}
			/*
			 * If both NORMAL_STEP and WATCH_STEP are in
			 * effect, give precedence to WATCH_STEP.
			 * One or the other must be set at this point.
			 */
			ASSERT(pcb->pcb_flags & (NORMAL_STEP|WATCH_STEP));
			if ((fault = undo_watch_step(&siginfo)) == 0 &&
			    (pcb->pcb_flags & NORMAL_STEP)) {
				siginfo.si_signo = SIGTRAP;
				siginfo.si_code = TRAP_TRACE;
				siginfo.si_addr = (caddr_t)rp->r_pc;
				fault = FLTTRACE;
			}
			pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
		}
		break;

	case T_DATA_EXCEPTION + T_USER: /* user data access exception */

		if (&vis1_partial_support != NULL) {
			bzero(&siginfo, sizeof (siginfo));
			if (vis1_partial_support(rp,
			    &siginfo, &fault) == 0)
				goto out;
		}

		if (nfload(rp, &instr))
			goto out;
		if (IS_FLUSH(instr)) {
			/* skip the flush instruction */
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
			goto out;
			/*NOTREACHED*/
		}
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_addr = addr;
		switch (X_FAULT_TYPE(mmu_fsr)) {
		case FT_ATOMIC_NC:
			if ((IS_SWAP(instr) && swap_nc(rp, instr)) ||
			    (IS_LDSTUB(instr) && ldstub_nc(rp, instr))) {
				/* skip the atomic */
```

```
                                rp->r_pc = rp->r_npc;
                                rp->r_npc += 4;
                                goto out;
                        }
                        /* fall into ... */
                case FT_PRIV:
                        siginfo.si_signo = SIGSEGV;
                        siginfo.si_code = SEGV_ACCERR;
                        fault = FLTBOUNDS;
                        break;
                case FT_SPEC_LD:
                case FT_ILL_ALT:
                        siginfo.si_signo = SIGILL;
                        siginfo.si_code = ILL_ILLADR;
                        fault = FLTILL;
                        break;
                default:
                        siginfo.si_signo = SIGSEGV;
                        siginfo.si_code = SEGV_MAPERR;
                        fault = FLTBOUNDS;
                        break;
                }
                break;

        case T_SYS_RTT_ALIGN + T_USER:  /* user alignment error */
        case T_ALIGNMENT + T_USER:      /* user alignment error */
                if (tudebug)
                        showregs(type, rp, addr, 0);
                /*
                 * If the user has to do unaligned references
                 * the ugly stuff gets done here.
                 */
                alignfaults++;
                if (&vis1_partial_support != NULL) {
                        bzero(&siginfo, sizeof (siginfo));
                        if (vis1_partial_support(rp,
                            &siginfo, &fault) == 0)
                                goto out;
                }

                bzero(&siginfo, sizeof (siginfo));
                if (type == T_SYS_RTT_ALIGN + T_USER) {
                        if (nfload(rp, NULL))
                                goto out;
                        /*
                         * Can't do unaligned stack access
                         */
                        siginfo.si_signo = SIGBUS;
                        siginfo.si_code = BUS_ADRALN;
                        siginfo.si_addr = addr;
                        fault = FLTACCESS;
                        break;
                }

                /*
                 * Try to fix alignment before non-faulting load test.
                 */
                if (p->p_fixalignment) {
                        if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
                                rp->r_pc = rp->r_npc;
                                rp->r_npc += 4;
                                goto out;
                        }
                        if (nfload(rp, NULL))
                                goto out;
                        siginfo.si_signo = SIGSEGV;
```

```
                        siginfo.si_code = SEGV_MAPERR;
                        siginfo.si_addr = badaddr;
                        fault = FLTBOUNDS;
                } else {
                        if (nfload(rp, NULL))
                                goto out;
                        siginfo.si_signo = SIGBUS;
                        siginfo.si_code = BUS_ADRALN;
                        if (rp->r_pc & 3) {   /* offending address, if pc */
                                siginfo.si_addr = (caddr_t)rp->r_pc;
                        } else {
                                if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
                                        siginfo.si_addr = badaddr;
                                else
                                        siginfo.si_addr = (caddr_t)rp->r_pc;
                        }
                        fault = FLTACCESS;
                }
                break;

        case T_PRIV_INSTR + T_USER:     /* privileged instruction fault */
                if (tudebug)
                        showregs(type, rp, (caddr_t)0, 0);

                bzero(&siginfo, sizeof (siginfo));
#ifdef  sun4v
                /*
                 * If this instruction fault is a non-privileged %tick
                 * or %stick trap, and %tick/%stick user emulation is
                 * enabled as a result of an OS suspend, then simulate
                 * the register read. We rely on simulate_rdtick to fail
                 * if the instruction is not a %tick or %stick read,
                 * causing us to fall through to the normal privileged
                 * instruction handling.
                 */
                if (tick_stick_emulation_active &&
                    (X_FAULT_TYPE(mmu_fsr) == FT_NEW_PRVACT) &&
                    simulate_rdtick(rp) == SIMU_SUCCESS) {
                        /* skip the successfully simulated instruction */
                        rp->r_pc = rp->r_npc;
                        rp->r_npc += 4;
                        goto out;
                }
#endif
                siginfo.si_signo = SIGILL;
                siginfo.si_code = ILL_PRVOPC;
                siginfo.si_addr = (caddr_t)rp->r_pc;
                fault = FLTILL;
                break;

        case T_UNIMP_INSTR:             /* priv illegal instruction fault */
                if (fpras_implemented) {
                        /*
                         * Call fpras_chktrap indicating that
                         * we've come from a trap handler and pass
                         * the regs.  That function may choose to panic
                         * (in which case it won't return) or it may
                         * determine that a reboot is desired.  In the
                         * latter case it must alter pc/npc to skip
                         * the illegal instruction and continue at
                         * a controlled address.
                         */
                        if (&fpras_chktrap) {
                                if (fpras_chktrap(rp))
                                        goto cleanup;
                        }
```

```
		}
#if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
		instr = *(int *)rp->r_pc;
		if ((instr & 0xc0000000) == 0x40000000) {
			long pc;

			rp->r_o7 = (long long)rp->r_pc;
			pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
			rp->r_pc = rp->r_npc;
			rp->r_npc = pc;
			ill_calls++;
			goto cleanup;
		}
#endif /* SF_ERRATA_23 || SF_ERRATA_30 */
		/*
		 * It's not an fpras failure and it's not SF_ERRATA_23 - die
		 */
		addr = (caddr_t)rp->r_pc;
		(void) die(type, rp, addr, 0);
		/*NOTREACHED*/

	case T_UNIMP_INSTR + T_USER:	/* illegal instruction fault */
#if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
		instr = fetch_user_instr((caddr_t)rp->r_pc);
		if ((instr & 0xc0000000) == 0x40000000) {
			long pc;

			rp->r_o7 = (long long)rp->r_pc;
			pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
			rp->r_pc = rp->r_npc;
			rp->r_npc = pc;
			ill_calls++;
			goto out;
		}
#endif /* SF_ERRATA_23 || SF_ERRATA_30 */
		if (tudebug)
			showregs(type, rp, (caddr_t)0, 0);
		bzero(&siginfo, sizeof (siginfo));
		/*
		 * Try to simulate the instruction.
		 */
		switch (simulate_unimp(rp, &badaddr)) {
		case SIMU_RETRY:
			goto out;	/* regs are already set up */
			/*NOTREACHED*/

		case SIMU_SUCCESS:
			/* skip the successfully simulated instruction */
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
			goto out;
			/*NOTREACHED*/

		case SIMU_FAULT:
			siginfo.si_signo = SIGSEGV;
			siginfo.si_code = SEGV_MAPERR;
			siginfo.si_addr = badaddr;
			fault = FLTBOUNDS;
			break;

		case SIMU_DZERO:
			siginfo.si_signo = SIGFPE;
			siginfo.si_code = FPE_INTDIV;
			siginfo.si_addr = (caddr_t)rp->r_pc;
			fault = FLTIZDIV;
			break;
```

```
		case SIMU_UNALIGN:
			siginfo.si_signo = SIGBUS;
			siginfo.si_code = BUS_ADRALN;
			siginfo.si_addr = badaddr;
			fault = FLTACCESS;
			break;

		case SIMU_ILLEGAL:
		default:
			siginfo.si_signo = SIGILL;
			op3 = (instr >> 19) & 0x3F;
			if ((IS_FLOAT(instr) && (op3 == IOP_V8_STQFA) ||
			    (op3 == IOP_V8_STDFA)))
				siginfo.si_code = ILL_ILLADR;
			else
				siginfo.si_code = ILL_ILLOPC;
			siginfo.si_addr = (caddr_t)rp->r_pc;
			fault = FLTILL;
			break;
		}
		break;

	case T_UNIMP_LDD + T_USER:
	case T_UNIMP_STD + T_USER:
		if (tudebug)
			showregs(type, rp, (caddr_t)0, 0);
		switch (simulate_lddstd(rp, &badaddr)) {
		case SIMU_SUCCESS:
			/* skip the successfully simulated instruction */
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
			goto out;
			/*NOTREACHED*/

		case SIMU_FAULT:
			if (nfload(rp, NULL))
				goto out;
			siginfo.si_signo = SIGSEGV;
			siginfo.si_code = SEGV_MAPERR;
			siginfo.si_addr = badaddr;
			fault = FLTBOUNDS;
			break;

		case SIMU_UNALIGN:
			if (nfload(rp, NULL))
				goto out;
			siginfo.si_signo = SIGBUS;
			siginfo.si_code = BUS_ADRALN;
			siginfo.si_addr = badaddr;
			fault = FLTACCESS;
			break;

		case SIMU_ILLEGAL:
		default:
			siginfo.si_signo = SIGILL;
			siginfo.si_code = ILL_ILLOPC;
			siginfo.si_addr = (caddr_t)rp->r_pc;
			fault = FLTILL;
			break;
		}
		break;

	case T_UNIMP_LDD:
	case T_UNIMP_STD:
		if (simulate_lddstd(rp, &badaddr) == SIMU_SUCCESS) {
			/* skip the successfully simulated instruction */
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
			goto cleanup;
			/*NOTREACHED*/
		}
		/*
		 * A third party driver executed an {LDD,STD,LDDA,STDA}
		 * that we couldn't simulate.
		 */
		if (nfload(rp, NULL))
			goto cleanup;

		if (curthread->t_lofault) {
			if (lodebug) {
				showregs(type, rp, addr, 0);
				traceback((caddr_t)rp->r_sp);
			}
			rp->r_g1 = EFAULT;
			rp->r_pc = curthread->t_lofault;
			rp->r_npc = rp->r_pc + 4;
			goto cleanup;
		}
		(void) die(type, rp, addr, 0);
		/*NOTREACHED*/

	case T_IDIV0 + T_USER:		/* integer divide by zero */
	case T_DIV0 + T_USER:		/* integer divide by zero */
		if (tudebug && tudebugfpe)
			showregs(type, rp, (caddr_t)0, 0);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_signo = SIGFPE;
		siginfo.si_code = FPE_INTDIV;
		siginfo.si_addr = (caddr_t)rp->r_pc;
		fault = FLTIZDIV;
		break;

	case T_INT_OVERFLOW + T_USER:	/* integer overflow */
		if (tudebug && tudebugfpe)
			showregs(type, rp, (caddr_t)0, 0);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_signo = SIGFPE;
		siginfo.si_code  = FPE_INTOVF;
		siginfo.si_addr  = (caddr_t)rp->r_pc;
		fault = FLTIOVF;
		break;

	case T_BREAKPOINT + T_USER:	/* breakpoint trap (t 1) */
		if (tudebug && tudebugbpt)
			showregs(type, rp, (caddr_t)0, 0);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_signo = SIGTRAP;
		siginfo.si_code = TRAP_BRKPT;
		siginfo.si_addr = (caddr_t)rp->r_pc;
		fault = FLTBPT;
		break;

	case T_TAG_OVERFLOW + T_USER:	/* tag overflow (taddcctv, tsubcctv) */
		if (tudebug)
			showregs(type, rp, (caddr_t)0, 0);
		bzero(&siginfo, sizeof (siginfo));
		siginfo.si_signo = SIGEMT;
		siginfo.si_code = EMT_TAGOVF;
		siginfo.si_addr = (caddr_t)rp->r_pc;
		fault = FLTACCESS;
		break;
```

```
	case T_FLUSH_PCB + T_USER:	/* finish user window overflow */
	case T_FLUSHW + T_USER:		/* finish user window flush */
		/*
		 * This trap is entered from sys_rtt in locore.s when,
		 * upon return to user is is found that there are user
		 * windows in pcb_wbuf.  This happens because they could
		 * not be saved on the user stack, either because it
		 * wasn't resident or because it was misaligned.
		 */
	{
		int error;
		caddr_t sp;

		error = flush_user_windows_to_stack(&sp);
		/*
		 * Possible errors:
		 *	error copying out
		 *	unaligned stack pointer
		 * The first is given to us as the return value
		 * from flush_user_windows_to_stack().  The second
		 * results in residual windows in the pcb.
		 */
		if (error != 0) {
			/*
			 * EINTR comes from a signal during copyout;
			 * we should not post another signal.
			 */
			if (error != EINTR) {
				/*
				 * Zap the process with a SIGSEGV - process
				 * may be managing its own stack growth by
				 * taking SIGSEGVs on a different signal stack.
				 */
				bzero(&siginfo, sizeof (siginfo));
				siginfo.si_signo = SIGSEGV;
				siginfo.si_code  = SEGV_MAPERR;
				siginfo.si_addr  = sp;
				fault = FLTBOUNDS;
			}
			break;
		} else if (mpcb->mpcb_wbcnt) {
			bzero(&siginfo, sizeof (siginfo));
			siginfo.si_signo = SIGILL;
			siginfo.si_code  = ILL_BADSTK;
			siginfo.si_addr  = (caddr_t)rp->r_pc;
			fault = FLTILL;
			break;
		}
	}

		/*
		 * T_FLUSHW is used when handling a ta 0x3 -- the old flush
		 * window trap -- which is implemented by executing the
		 * flushw instruction. The flushw can trap if any of the
		 * stack pages are not writable for whatever reason. In this
		 * case only, we advance the pc to the next instruction so
		 * that the user thread doesn't needlessly execute the trap
		 * again. Normally this wouldn't be a problem -- we'll
		 * usually only end up here if this is the first touch to a
		 * stack page -- since the second execution won't trap, but
		 * if there's a watchpoint on the stack page the user thread
		 * would spin, continuously executing the trap instruction.
		 */
		if (type == T_FLUSHW + T_USER) {
			rp->r_pc = rp->r_npc;
```

```
                        rp->r_npc += 4;
                }
                goto out;

        case T_AST + T_USER:            /* profiling or resched pseudo trap */
                if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
                        lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
                        if (kcpc_overflow_ast()) {
                                /*
                                 * Signal performance counter overflow
                                 */
                                if (tudebug)
                                        showregs(type, rp, (caddr_t)0, 0);
                                bzero(&siginfo, sizeof (siginfo));
                                siginfo.si_signo = SIGEMT;
                                siginfo.si_code = EMT_CPCOVF;
                                siginfo.si_addr = (caddr_t)rp->r_pc;
                                /* for trap_cleanup(), below */
                                oldpc = rp->r_pc - 4;
                                fault = FLTCPCOVF;
                        }
                }

                /*
                 * The CPC_OVERFLOW check above may already have populated
                 * siginfo and set fault, so the checks below must not
                 * touch these and the functions they call must use
                 * trapsig() directly.
                 */

                if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
                        trap_async_hwerr();
                }

                if (lwp->lwp_pcb.pcb_flags & ASYNC_BERR) {
                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_BERR;
                        trap_async_berr_bto(ASYNC_BERR, rp);
                }

                if (lwp->lwp_pcb.pcb_flags & ASYNC_BTO) {
                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_BTO;
                        trap_async_berr_bto(ASYNC_BTO, rp);
                }

                break;
        }

        if (fault) {
                /* We took a fault so abort single step. */
                lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
        }
        trap_cleanup(rp, fault, &siginfo, oldpc == rp->r_pc);

out:    /* We can't get here from a system trap */
        ASSERT(type & T_USER);
        trap_rtt();
        (void) new_mstate(curthread, mstate);
        /* Kernel probe */
        TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
                tnf_microstate, state, LMS_USER);

        TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
        return;

cleanup:        /* system traps end up here */
```

```
        ASSERT(!(type & T_USER));

        TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
}
______unchanged_portion_omitted_

/*
 * Called from fp_traps when a floating point trap occurs.
 * Note that the T_DATA_EXCEPTION case does not use X_FAULT_TYPE(mmu_fsr),
 * because mmu_fsr (now changed to code) is always 0.
 * Note that the T_UNIMP_INSTR case does not call simulate_unimp(),
 * because the simulator only simulates multiply and divide instructions,
 * which would not cause floating point traps in the first place.
 * XXX - Supervisor mode floating point traps?
 */
void
fpu_trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t code)
{
        proc_t *p = ttoproc(curthread);
        klwp_id_t lwp = ttolwp(curthread);
        k_siginfo_t siginfo;
        uint_t op3, fault = 0;
        int mstate;
        char *badaddr;
        kfpu_t *fp;
        struct _fpq *pfpq;
        uint32_t inst;
        utrap_handler_t *utrapp;

        CPU_STATS_ADDQ(CPU, sys, trap, 1);

        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

        if (USERMODE(rp->r_tstate)) {
                /*
                 * Set lwp_state before trying to acquire any
                 * adaptive lock
                 */
                ASSERT(lwp != NULL);
                lwp->lwp_state = LWP_SYS;
                /*
                 * Set up the current cred to use during this trap. u_cred
                 * no longer exists.  t_cred is used instead.
                 * The current process credential applies to the thread for
                 * the entire trap.  If trapping from the kernel, this
                 * should already be set up.
                 */
                if (curthread->t_cred != p->p_cred) {
                        cred_t *oldcred = curthread->t_cred;
                        /*
                         * DTrace accesses t_cred in probe context.  t_cred
                         * must always be either NULL, or point to a valid,
                         * allocated cred structure.
                         */
                        curthread->t_cred = crgetcred();
                        crfree(oldcred);
                }
                ASSERT(lwp->lwp_regs == rp);
                mstate = new_mstate(curthread, LMS_TRAP);
                siginfo.si_signo = 0;
                type |= T_USER;
        }

        TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
            "C_fpu_trap_handler_enter:type %x", type);
	if (tudebug && tudebugfpe)
		showregs(type, rp, addr, 0);

	bzero(&siginfo, sizeof (siginfo));
	siginfo.si_code = code;
	siginfo.si_addr = addr;

	switch (type) {

	case T_FP_EXCEPTION_IEEE + T_USER:	/* FPU arithmetic exception */
		/*
		 * FPU arithmetic exception - fake up a fpq if we
		 *	came here directly from _fp_ieee_exception,
		 *	which is indicated by a zero fpu_qcnt.
		 */
		fp = lwptofpu(curthread->t_lwp);
		utrapp = curthread->t_procp->p_utraps;
		if (fp->fpu_qcnt == 0) {
			inst = fetch_user_instr((caddr_t)rp->r_pc);
			lwp->lwp_state = LWP_SYS;
			pfpq = &fp->fpu_q->FQu.fpq;
			pfpq->fpq_addr = (uint32_t *)rp->r_pc;
			pfpq->fpq_instr = inst;
			fp->fpu_qcnt = 1;
			fp->fpu_q_entrysize = sizeof (struct _fpq);
#ifdef SF_V9_TABLE_28
			/*
			 * Spitfire and blackbird followed the SPARC V9 manual
			 * paragraph 3 of section 5.1.7.9 FSR_current_exception
			 * (cexc) for setting fsr.cexc bits on underflow and
			 * overflow traps when the fsr.tem.inexact bit is set,
			 * instead of following Table 28. Bugid 1263234.
			 */
			{
				extern int spitfire_bb_fsr_bug;

				if (spitfire_bb_fsr_bug &&
				    (fp->fpu_fsr & FSR_TEM_NX)) {
					if (((fp->fpu_fsr & FSR_TEM_OF) == 0) &&
					    (fp->fpu_fsr & FSR_CEXC_OF)) {
						fp->fpu_fsr &= ~FSR_CEXC_OF;
						fp->fpu_fsr |= FSR_CEXC_NX;
						_fp_write_pfsr(&fp->fpu_fsr);
						siginfo.si_code = FPE_FLTRES;
					}
					if (((fp->fpu_fsr & FSR_TEM_UF) == 0) &&
					    (fp->fpu_fsr & FSR_CEXC_UF)) {
						fp->fpu_fsr &= ~FSR_CEXC_UF;
						fp->fpu_fsr |= FSR_CEXC_NX;
						_fp_write_pfsr(&fp->fpu_fsr);
						siginfo.si_code = FPE_FLTRES;
					}
				}
			}
#endif /* SF_V9_TABLE_28 */
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
		} else if (utrapp && utrapp[UT_FP_EXCEPTION_IEEE_754]) {
			/*
			 * The user had a trap handler installed.  Jump to
			 * the trap handler instead of signalling the process.
			 */
			rp->r_pc = (long)utrapp[UT_FP_EXCEPTION_IEEE_754];
			rp->r_npc = rp->r_pc + 4;
			break;
		}
```

```
		siginfo.si_signo = SIGFPE;
		fault = FLTFPE;
		break;

	case T_DATA_EXCEPTION + T_USER:		/* user data access exception */
		siginfo.si_signo = SIGSEGV;
		fault = FLTBOUNDS;
		break;

	case T_LDDF_ALIGN + T_USER: /* 64 bit user lddfa alignment error */
	case T_STDF_ALIGN + T_USER: /* 64 bit user stdfa alignment error */
		alignfaults++;
		lwp->lwp_state = LWP_SYS;
		if (&vis1_partial_support != NULL) {
			bzero(&siginfo, sizeof (siginfo));
			if (vis1_partial_support(rp,
			    &siginfo, &fault) == 0)
				goto out;
		}
		if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
			rp->r_pc = rp->r_npc;
			rp->r_npc += 4;
			goto out;
		}
		fp = lwptofpu(curthread->t_lwp);
		fp->fpu_qcnt = 0;
		siginfo.si_signo = SIGSEGV;
		siginfo.si_code = SEGV_MAPERR;
		siginfo.si_addr = badaddr;
		fault = FLTBOUNDS;
		break;

	case T_ALIGNMENT + T_USER:		/* user alignment error */
		/*
		 * If the user has to do unaligned references
		 * the ugly stuff gets done here.
		 * Only handles vanilla loads and stores.
		 */
		alignfaults++;
		if (p->p_fixalignment) {
			if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
				rp->r_pc = rp->r_npc;
				rp->r_npc += 4;
				goto out;
			}
			siginfo.si_signo = SIGSEGV;
			siginfo.si_code = SEGV_MAPERR;
			siginfo.si_addr = badaddr;
			fault = FLTBOUNDS;
		} else {
			siginfo.si_signo = SIGBUS;
			siginfo.si_code = BUS_ADRALN;
			if (rp->r_pc & 3) {	/* offending address, if pc */
				siginfo.si_addr = (caddr_t)rp->r_pc;
			} else {
				if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
					siginfo.si_addr = badaddr;
				else
					siginfo.si_addr = (caddr_t)rp->r_pc;
			}
			fault = FLTACCESS;
		}
		break;

	case T_UNIMP_INSTR + T_USER:		/* illegal instruction fault */
		siginfo.si_signo = SIGILL;
```

```
                inst = fetch_user_instr((caddr_t)rp->r_pc);
                op3 = (inst >> 19) & 0x3F;
                if ((op3 == IOP_V8_STQFA) || (op3 == IOP_V8_STDFA))
                        siginfo.si_code = ILL_ILLADR;
                else
                        siginfo.si_code = ILL_ILLTRP;
                fault = FLTILL;
                break;

        default:
                (void) die(type, rp, addr, 0);
                /*NOTREACHED*/
        }

        /*
         * We can't get here from a system trap
         * Never restart any instruction which got here from an fp trap.
         */
        ASSERT(type & T_USER);

        trap_cleanup(rp, fault, &siginfo, 0);
out:
        trap_rtt();
        (void) new_mstate(curthread, mstate);
}
```
______unchanged_portion_omitted_