```
**********************************************************
  187256 Fri Jul 17 10:39:33 2015
new/usr/src/uts/common/vm/vm_page.c
6065 page hash: use a static inline instead of a macro
**********************************************************
```

```
/*
 * 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) 2015, Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

/*      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.
 */

/*
 * VM - physical page management.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vm.h>
#include <sys/vtrace.h>
#include <sys/swap.h>
#include <sys/cmn_err.h>
#include <sys/tuneable.h>
#include <sys/sysmacros.h>
#include <sys/cpuvar.h>
#include <sys/callb.h>
#include <sys/debug.h>
#include <sys/tnf_probe.h>
#include <sys/condvar_impl.h>
```



```
#include <sys/mem_config.h>
#include <sys/mem_cage.h>
#include <sys/kmem.h>
#include <sys/atomic.h>
#include <sys/strlog.h>
#include <sys/mman.h>
#include <sys/ontrap.h>
#include <sys/lgrp.h>
#include <sys/vfs.h>

#include <vm/hat.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/seg.h>
#include <vm/pvn.h>
#include <vm/seg_kmem.h>
#include <vm/vm_dep.h>
#include <sys/vm_usage.h>
#include <fs/fs_subr.h>
#include <sys/ddi.h>
#include <sys/modctl.h>

static pgcnt_t max_page_get;    /* max page_get request size in pages */
pgcnt_t total_pages = 0;        /* total number of pages (used by /proc) */

/*
 * freemem_lock protects all freemem variables:
 * availrmem. Also this lock protects the globals which track the
 * availrmem changes for accurate kernel footprint calculation.
 * See below for an explanation of these
 * globals.
 */
kmutex_t freemem_lock;
pgcnt_t availrmem;
pgcnt_t availrmem_initial;

/*
 * These globals track availrmem changes to get a more accurate
 * estimate of tke kernel size. Historically pp_kernel is used for
 * kernel size and is based on availrmem. But availrmem is adjusted for
 * locked pages in the system not just for kernel locked pages.
 * These new counters will track the pages locked through segvn and
 * by explicit user locking.
 *
 * pages_locked : How many pages are locked because of user specified
 * locking through mlock or plock.
 *
 * pages_useclaim,pages_claimed : These two variables track the
 * claim adjustments because of the protection changes on a segvn segment.
 *
 * All these globals are protected by the same lock which protects availrmem.
 */
pgcnt_t pages_locked = 0;
pgcnt_t pages_useclaim = 0;
pgcnt_t pages_claimed = 0;


/*
 * new_freemem_lock protects freemem, freemem_wait & freemem_cv.
 */
static kmutex_t new_freemem_lock;
static uint_t   freemem_wait;   /* someone waiting for freemem */
static kcondvar_t freemem_cv;

/*
 * The logical page free list is maintained as two lists, the 'free'
```

```
 * and the 'cache' lists.
 * The free list contains those pages that should be reused first.
 *
 * The implementation of the lists is machine dependent.
 * page_get_freelist(), page_get_cachelist(),
 * page_list_sub(), and page_list_add()
 * form the interface to the machine dependent implementation.
 *
 * Pages with p_free set are on the cache list.
 * Pages with p_free and p_age set are on the free list,
 *
 * A page may be locked while on either list.
 */

/*
 * free list accounting stuff.
 *
 *
 * Spread out the value for the number of pages on the
 * page free and page cache lists.  If there is just one
 * value, then it must be under just one lock.
 * The lock contention and cache traffic are a real bother.
 *
 * When we acquire and then drop a single pcf lock
 * we can start in the middle of the array of pcf structures.
 * If we acquire more than one pcf lock at a time, we need to
 * start at the front to avoid deadlocking.
 *
 * pcf_count holds the number of pages in each pool.
 *
 * pcf_block is set when page_create_get_something() has asked the
 * PSM page freelist and page cachelist routines without specifying
 * a color and nothing came back.  This is used to block anything
 * else from moving pages from one list to the other while the
 * lists are searched again.  If a page is freeed while pcf_block is
 * set, then pcf_reserve is incremented.  pcgs_unblock() takes care
 * of clearning pcf_block, doing the wakeups, etc.
 */

#define MAX_PCF_FANOUT NCPU
static uint_t pcf_fanout = 1; /* Will get changed at boot time */
static uint_t pcf_fanout_mask = 0;

struct pcf {
        kmutex_t        pcf_lock;       /* protects the structure */
        uint_t          pcf_count;      /* page count */
        uint_t          pcf_wait;       /* number of waiters */
        uint_t          pcf_block;      /* pcgs flag to page_free() */
        uint_t          pcf_reserve;    /* pages freed after pcf_block set */
        uint_t          pcf_fill[10];   /* to line up on the caches */
};

/*
 * PCF_INDEX hash needs to be dynamic (every so often the hash changes where
 * it will hash the cpu to).  This is done to prevent a drain condition
 * from happening.  This drain condition will occur when pcf_count decrement
 * occurs on cpu A and the increment of pcf_count always occurs on cpu B.  An
 * example of this shows up with device interrupts.  The dma buffer is allocated
 * by the cpu requesting the IO thus the pcf_count is decremented based on that.
 * When the memory is returned by the interrupt thread, the pcf_count will be
 * incremented based on the cpu servicing the interrupt.
 */
static struct pcf pcf[MAX_PCF_FANOUT];
#define PCF_INDEX() ((int)(((long)CPU->cpu_seqid) + \
        (randtick() >> 24)) & (pcf_fanout_mask))
```



```
static int pcf_decrement_bucket(pgcnt_t);
static int pcf_decrement_multiple(pgcnt_t *, pgcnt_t, int);

kmutex_t        pcgs_lock;              /* serializes page_create_get_ */
kmutex_t        pcgs_cagelock;          /* serializes NOSLEEP cage allocs */
kmutex_t        pcgs_wait_lock;         /* used for delay in pcgs */
static kcondvar_t       pcgs_cv;        /* cv for delay in pcgs */

#ifdef VM_STATS

/*
 * No locks, but so what, they are only statistics.
 */

static struct page_tcnt {
        int     pc_free_cache;          /* free's into cache list */
        int     pc_free_dontneed;       /* free's with dontneed */
        int     pc_free_pageout;        /* free's from pageout */
        int     pc_free_free;           /* free's into free list */
        int     pc_free_pages;          /* free's into large page free list */
        int     pc_destroy_pages;       /* large page destroy's */
        int     pc_get_cache;           /* get's from cache list */
        int     pc_get_free;            /* get's from free list */
        int     pc_reclaim;             /* reclaim's */
        int     pc_abortfree;           /* abort's of free pages */
        int     pc_find_hit;            /* find's that find page */
        int     pc_find_miss;           /* find's that don't find page */
        int     pc_destroy_free;        /* # of free pages destroyed */
#define PC_HASH_CNT     (4*PAGE_HASHAVELEN)
        int     pc_find_hashlen[PC_HASH_CNT+1];
        int     pc_addclaim_pages;
        int     pc_subclaim_pages;
        int     pc_free_replacement_page[2];
        int     pc_try_demote_pages[6];
        int     pc_demote_pages[2];
} pagecnt;

uint_t  hashin_count;
uint_t  hashin_not_held;
uint_t  hashin_already;

uint_t  hashout_count;
uint_t  hashout_not_held;

uint_t  page_create_count;
uint_t  page_create_not_enough;
uint_t  page_create_not_enough_again;
uint_t  page_create_zero;
uint_t  page_create_hashout;
uint_t  page_create_page_lock_failed;
uint_t  page_create_trylock_failed;
uint_t  page_create_found_one;
uint_t  page_create_hashin_failed;
uint_t  page_create_dropped_phm;

uint_t  page_create_new;
uint_t  page_create_exists;
uint_t  page_create_putbacks;
uint_t  page_create_overshoot;

uint_t  page_reclaim_zero;
uint_t  page_reclaim_zero_locked;

uint_t  page_rename_exists;
uint_t  page_rename_count;
```

```
260 uint_t  page_lookup_cnt[20];
261 uint_t  page_lookup_nowait_cnt[10];
262 uint_t  page_find_cnt;
263 uint_t  page_exists_cnt;
264 uint_t  page_exists_forreal_cnt;
265 uint_t  page_lookup_dev_cnt;
266 uint_t  get_cachelist_cnt;
267 uint_t  page_create_cnt[10];
268 uint_t  alloc_pages[9];
269 uint_t  page_exphcontg[19];
270 uint_t  page_create_large_cnt[10];

272 #endif

274 static inline page_t *
275 page_hash_search(ulong_t index, vnode_t *vnode, u_offset_t off)
276 {
277         uint_t mylen = 0;
278         page_t *page;

280         for (page = page_hash[index]; page; page = page->p_hash, mylen++)
281                 if (page->p_vnode == vnode && page->p_offset == off)
282                         break;

284 #ifdef  VM_STATS
285         if (page != NULL)
286                 pagecnt.pc_find_hit++;
287         else
288                 pagecnt.pc_find_miss++;
 23 /*
 24  * Collects statistics.
 25  */
 26 #define PAGE_HASH_SEARCH(index, pp, vp, off) { \
 27         uint_t  mylen = 0; \
 28                 \
 29         for ((pp) = page_hash[(index)]; (pp); (pp) = (pp)->p_hash, mylen++) { \
 30                 if ((pp)->p_vnode == (vp) && (pp)->p_offset == (off)) \
 31                         break; \
 32         } \
 33         if ((pp) != NULL) \
 34                 pagecnt.pc_find_hit++; \
 35         else \
 36                 pagecnt.pc_find_miss++; \
 37         if (mylen > PC_HASH_CNT) \
 38                 mylen = PC_HASH_CNT; \
 39         pagecnt.pc_find_hashlen[mylen]++; \
 40 }

 42 #else   /* VM_STATS */

 44 /*
 45  * Don't collect statistics
 46  */
 47 #define PAGE_HASH_SEARCH(index, pp, vp, off) { \
 48         for ((pp) = page_hash[(index)]; (pp); (pp) = (pp)->p_hash) { \
 49                 if ((pp)->p_vnode == (vp) && (pp)->p_offset == (off)) \
 50                         break; \
 51         } \
 52 }

290         pagecnt.pc_find_hashlen[MIN(mylen, PC_HASH_CNT)]++;
291 #endif
 54 #endif  /* VM_STATS */

293         return (page);
294 }
```



```
295 #endif /* ! codereview */


298 #ifdef DEBUG
299 #define MEMSEG_SEARCH_STATS
300 #endif

302 #ifdef MEMSEG_SEARCH_STATS
303 struct memseg_stats {
304     uint_t nsearch;
305     uint_t nlastwon;
306     uint_t nhashwon;
307     uint_t nnotfound;
308 } memseg_stats;

310 #define MEMSEG_STAT_INCR(v) \
311         atomic_inc_32(&memseg_stats.v)
312 #else
313 #define MEMSEG_STAT_INCR(x)
314 #endif

316 struct memseg *memsegs;         /* list of memory segments */

318 /*
319  * /etc/system tunable to control large page allocation hueristic.
320  *
321  * Setting to LPAP_LOCAL will heavily prefer the local lgroup over remote lgroup
322  * for large page allocation requests.  If a large page is not readily
323  * avaliable on the local freelists we will go through additional effort
324  * to create a large page, potentially moving smaller pages around to coalesce
325  * larger pages in the local lgroup.
326  * Default value of LPAP_DEFAULT will go to remote freelists if large pages
327  * are not readily available in the local lgroup.
328  */
329 enum lpap {
330         LPAP_DEFAULT,   /* default large page allocation policy */
331         LPAP_LOCAL      /* local large page allocation policy */
332 };

334 enum lpap lpg_alloc_prefer = LPAP_DEFAULT;

336 static void page_init_mem_config(void);
337 static int page_do_hashin(page_t *, vnode_t *, u_offset_t);
338 static void page_do_hashout(page_t *);
339 static void page_capture_init();
340 int page_capture_take_action(page_t *, uint_t, void *);

342 static void page_demote_vp_pages(page_t *);


345 void
346 pcf_init(void)

348 {
349         if (boot_ncpus != -1) {
350                 pcf_fanout = boot_ncpus;
351         } else {
352                 pcf_fanout = max_ncpus;
353         }
354 #ifdef sun4v
355         /*
356          * Force at least 4 buckets if possible for sun4v.
357          */
358         pcf_fanout = MAX(pcf_fanout, 4);
359 #endif /* sun4v */
```

```
361         /*
362          * Round up to the nearest power of 2.
363          */
364         pcf_fanout = MIN(pcf_fanout, MAX_PCF_FANOUT);
365         if (!ISP2(pcf_fanout)) {
366                 pcf_fanout = 1 << highbit(pcf_fanout);

368                 if (pcf_fanout > MAX_PCF_FANOUT) {
369                         pcf_fanout = 1 << (highbit(MAX_PCF_FANOUT) - 1);
370                 }
371         }
372         pcf_fanout_mask = pcf_fanout - 1;
373 }

375 /*
376  * vm subsystem related initialization
377  */
378 void
379 vm_init(void)
380 {
381         boolean_t callb_vm_cpr(void *, int);

383         (void) callb_add(callb_vm_cpr, 0, CB_CL_CPR_VM, "vm");
384         page_init_mem_config();
385         page_retire_init();
386         vm_usage_init();
387         page_capture_init();
388 }

390 /*
391  * This function is called at startup and when memory is added or deleted.
392  */
393 void
394 init_pages_pp_maximum()
395 {
396         static pgcnt_t p_min;
397         static pgcnt_t pages_pp_maximum_startup;
398         static pgcnt_t avrmem_delta;
399         static int init_done;
400         static int user_set;    /* true if set in /etc/system */

402         if (init_done == 0) {

404                 /* If the user specified a value, save it */
405                 if (pages_pp_maximum != 0) {
406                         user_set = 1;
407                         pages_pp_maximum_startup = pages_pp_maximum;
408                 }

410                 /*
411                  * Setting of pages_pp_maximum is based first time
412                  * on the value of availrmem just after the start-up
413                  * allocations. To preserve this relationship at run
414                  * time, use a delta from availrmem_initial.
415                  */
416                 ASSERT(availrmem_initial >= availrmem);
417                 avrmem_delta = availrmem_initial - availrmem;

419                 /* The allowable floor of pages_pp_maximum */
420                 p_min = tune.t_minarmem + 100;

422                 /* Make sure we don't come through here again. */
423                 init_done = 1;
424         }
425         /*
426          * Determine pages_pp_maximum, the number of currently available
```



```
427          * pages (availrmem) that can't be 'locked'. If not set by
428          * the user, we set it to 4% of the currently available memory
429          * plus 4MB.
430          * But we also insist that it be greater than tune.t_minarmem;
431          * otherwise a process could lock down a lot of memory, get swapped
432          * out, and never have enough to get swapped back in.
433          */
434         if (user_set)
435                 pages_pp_maximum = pages_pp_maximum_startup;
436         else
437                 pages_pp_maximum = ((availrmem_initial - avrmem_delta) / 25)
438                     + btop(4 * 1024 * 1024);

440         if (pages_pp_maximum <= p_min) {
441                 pages_pp_maximum = p_min;
442         }
443 }

445 void
446 set_max_page_get(pgcnt_t target_total_pages)
447 {
448         max_page_get = target_total_pages / 2;
449 }

451 static pgcnt_t pending_delete;

453 /*ARGSUSED*/
454 static void
455 page_mem_config_post_add(
456         void *arg,
457         pgcnt_t delta_pages)
458 {
459         set_max_page_get(total_pages - pending_delete);
460         init_pages_pp_maximum();
461 }

463 /*ARGSUSED*/
464 static int
465 page_mem_config_pre_del(
466         void *arg,
467         pgcnt_t delta_pages)
468 {
469         pgcnt_t nv;

471         nv = atomic_add_long_nv(&pending_delete, (spgcnt_t)delta_pages);
472         set_max_page_get(total_pages - nv);
473         return (0);
474 }

476 /*ARGSUSED*/
477 static void
478 page_mem_config_post_del(
479         void *arg,
480         pgcnt_t delta_pages,
481         int cancelled)
482 {
483         pgcnt_t nv;

485         nv = atomic_add_long_nv(&pending_delete, -(spgcnt_t)delta_pages);
486         set_max_page_get(total_pages - nv);
487         if (!cancelled)
488                 init_pages_pp_maximum();
489 }

491 static kphysm_setup_vector_t page_mem_config_vec = {
492         KPHYSM_SETUP_VECTOR_VERSION,
```

```
493         page_mem_config_post_add,
494         page_mem_config_pre_del,
495         page_mem_config_post_del,
496 };

498 static void
499 page_init_mem_config(void)
500 {
501         int ret;

503         ret = kphysm_setup_func_register(&page_mem_config_vec, (void *)NULL);
504         ASSERT(ret == 0);
505 }

507 /*
508  * Evenly spread out the PCF counters for large free pages
509  */
510 static void
511 page_free_large_ctr(pgcnt_t npages)
512 {
513         static struct pcf       *p = pcf;
514         pgcnt_t                 lump;

516         freemem += npages;

518         lump = roundup(npages, pcf_fanout) / pcf_fanout;

520         while (npages > 0) {

522                 ASSERT(!p->pcf_block);

524                 if (lump < npages) {
525                         p->pcf_count += (uint_t)lump;
526                         npages -= lump;
527                 } else {
528                         p->pcf_count += (uint_t)npages;
529                         npages = 0;
530                 }

532                 ASSERT(!p->pcf_wait);

534                 if (++p > &pcf[pcf_fanout - 1])
535                         p = pcf;
536         }

538         ASSERT(npages == 0);
539 }

541 /*
542  * Add a physical chunk of memory to the system free lists during startup.
543  * Platform specific startup() allocates the memory for the page structs.
544  *
545  * num  - number of page structures
546  * base - page number (pfn) to be associated with the first page.
547  *
548  * Since we are doing this during startup (ie. single threaded), we will
549  * use shortcut routines to avoid any locking overhead while putting all
550  * these pages on the freelists.
551  *
552  * NOTE: Any changes performed to page_free(), must also be performed to
553  *       add_physmem() since this is how we initialize all page_t's at
554  *       boot time.
555  */
556 void
557 add_physmem(
558         page_t  *pp,
```



```
559         pgcnt_t num,
560         pfn_t   pnum)
561 {
562         page_t  *root = NULL;
563         uint_t  szc = page_num_pagesizes() - 1;
564         pgcnt_t large = page_get_pagecnt(szc);
565         pgcnt_t cnt = 0;

567         TRACE_2(TR_FAC_VM, TR_PAGE_INIT,
568             "add_physmem:pp %p num %lu", pp, num);

570         /*
571          * Arbitrarily limit the max page_get request
572          * to 1/2 of the page structs we have.
573          */
574         total_pages += num;
575         set_max_page_get(total_pages);

577         PLCNT_MODIFY_MAX(pnum, (long)num);

579         /*
580          * The physical space for the pages array
581          * representing ram pages has already been
582          * allocated.  Here we initialize each lock
583          * in the page structure, and put each on
584          * the free list
585          */
586         for (; num; pp++, pnum++, num--) {

588                 /*
589                  * this needs to fill in the page number
590                  * and do any other arch specific initialization
591                  */
592                 add_physmem_cb(pp, pnum);

594                 pp->p_lckcnt = 0;
595                 pp->p_cowcnt = 0;
596                 pp->p_slckcnt = 0;

598                 /*
599                  * Initialize the page lock as unlocked, since nobody
600                  * can see or access this page yet.
601                  */
602                 pp->p_selock = 0;

604                 /*
605                  * Initialize IO lock
606                  */
607                 page_iolock_init(pp);

609                 /*
610                  * initialize other fields in the page_t
611                  */
612                 PP_SETFREE(pp);
613                 page_clr_all_props(pp);
614                 PP_SETAGED(pp);
615                 pp->p_offset = (u_offset_t)-1;
616                 pp->p_next = pp;
617                 pp->p_prev = pp;

619                 /*
620                  * Simple case: System doesn't support large pages.
621                  */
622                 if (szc == 0) {
623                         pp->p_szc = 0;
624                         page_free_at_startup(pp);
```

```
625                         continue;
626                 }

628                 /*
629                  * Handle unaligned pages, we collect them up onto
630                  * the root page until we have a full large page.
631                  */
632                 if (!IS_P2ALIGNED(pnum, large)) {

634                         /*
635                          * If not in a large page,
636                          * just free as small page.
637                          */
638                         if (root == NULL) {
639                                 pp->p_szc = 0;
640                                 page_free_at_startup(pp);
641                                 continue;
642                         }

644                         /*
645                          * Link a constituent page into the large page.
646                          */
647                         pp->p_szc = szc;
648                         page_list_concat(&root, &pp);

650                         /*
651                          * When large page is fully formed, free it.
652                          */
653                         if (++cnt == large) {
654                                 page_free_large_ctr(cnt);
655                                 page_list_add_pages(root, PG_LIST_ISINIT);
656                                 root = NULL;
657                                 cnt = 0;
658                         }
659                         continue;
660                 }

662                 /*
663                  * At this point we have a page number which
664                  * is aligned. We assert that we aren't already
665                  * in a different large page.
666                  */
667                 ASSERT(IS_P2ALIGNED(pnum, large));
668                 ASSERT(root == NULL && cnt == 0);

670                 /*
671                  * If insufficient number of pages left to form
672                  * a large page, just free the small page.
673                  */
674                 if (num < large) {
675                         pp->p_szc = 0;
676                         page_free_at_startup(pp);
677                         continue;
678                 }

680                 /*
681                  * Otherwise start a new large page.
682                  */
683                 pp->p_szc = szc;
684                 cnt++;
685                 root = pp;
686         }
687         ASSERT(root == NULL && cnt == 0);
688 }

690 /*
```



```
691  * Find a page representing the specified [vp, offset].
692  * If we find the page but it is intransit coming in,
693  * it will have an "exclusive" lock and we wait for
694  * the i/o to complete.  A page found on the free list
695  * is always reclaimed and then locked.  On success, the page
696  * is locked, its data is valid and it isn't on the free
697  * list, while a NULL is returned if the page doesn't exist.
698  */
699 page_t *
700 page_lookup(vnode_t *vp, u_offset_t off, se_t se)
701 {
702         return (page_lookup_create(vp, off, se, NULL, NULL, 0));
703 }

705 /*
706  * Find a page representing the specified [vp, offset].
707  * We either return the one we found or, if passed in,
708  * create one with identity of [vp, offset] of the
709  * pre-allocated page. If we find existing page but it is
710  * intransit coming in, it will have an "exclusive" lock
711  * and we wait for the i/o to complete.  A page found on
712  * the free list is always reclaimed and then locked.
713  * On success, the page is locked, its data is valid and
714  * it isn't on the free list, while a NULL is returned
715  * if the page doesn't exist and newpp is NULL;
716  */
717 page_t *
718 page_lookup_create(
719         vnode_t *vp,
720         u_offset_t off,
721         se_t se,
722         page_t *newpp,
723         spgcnt_t *nrelocp,
724         int flags)
725 {
726         page_t          *pp;
727         kmutex_t        *phm;
728         ulong_t         index;
729         uint_t          hash_locked;
730         uint_t          es;

732         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
733         VM_STAT_ADD(page_lookup_cnt[0]);
734         ASSERT(newpp ? PAGE_EXCL(newpp) : 1);

736         /*
737          * Acquire the appropriate page hash lock since
738          * we have to search the hash list.  Pages that
739          * hash to this list can't change identity while
740          * this lock is held.
741          */
742         hash_locked = 0;
743         index = PAGE_HASH_FUNC(vp, off);
744         phm = NULL;
745 top:
746         pp = page_hash_search(index, vp, off);
 56         PAGE_HASH_SEARCH(index, pp, vp, off);
747         if (pp != NULL) {
748                 VM_STAT_ADD(page_lookup_cnt[1]);
749                 es = (newpp != NULL) ? 1 : 0;
750                 es |= flags;
751                 if (!hash_locked) {
752                         VM_STAT_ADD(page_lookup_cnt[2]);
753                         if (!page_try_reclaim_lock(pp, se, es)) {
754                                 /*
755                                  * On a miss, acquire the phm.  Then
```

```
 756                                 * next time, page_lock() will be called,
 757                                 * causing a wait if the page is busy.
 758                                 * just looping with page_trylock() would
 759                                 * get pretty boring.
 760                                 */
 761                                VM_STAT_ADD(page_lookup_cnt[3]);
 762                                phm = PAGE_HASH_MUTEX(index);
 763                                mutex_enter(phm);
 764                                hash_locked = 1;
 765                                goto top;
 766                        }
 767                } else {
 768                        VM_STAT_ADD(page_lookup_cnt[4]);
 769                        if (!page_lock_es(pp, se, phm, P_RECLAIM, es)) {
 770                                VM_STAT_ADD(page_lookup_cnt[5]);
 771                                goto top;
 772                        }
 773                }

 775                /*
 776                 * Since ‘pp’ is locked it can not change identity now.
 777                 * Reconfirm we locked the correct page.
 778                 *
 779                 * Both the p_vnode and p_offset *must* be cast volatile
 780                 * to force a reload of their values: The page_hash_search
 781                 * function will have stuffed p_vnode and p_offset into
  90                 * to force a reload of their values: The PAGE_HASH_SEARCH
  91                 * macro will have stuffed p_vnode and p_offset into
 782                 * registers before calling page_trylock(); another thread,
 783                 * actually holding the hash lock, could have changed the
 784                 * page’s identity in memory, but our registers would not
 785                 * be changed, fooling the reconfirmation.  If the hash
 786                 * lock was held during the search, the casting would
 787                 * not be needed.
 788                 */
 789                VM_STAT_ADD(page_lookup_cnt[6]);
 790                if (((volatile struct vnode *)(pp->p_vnode) != vp) ||
 791                    ((volatile u_offset_t)(pp->p_offset) != off)) {
 792                        VM_STAT_ADD(page_lookup_cnt[7]);
 793                        if (hash_locked) {
 794                                panic("page_lookup_create: lost page %p",
 795                                    (void *)pp);
 796                                /*NOTREACHED*/
 797                        }
 798                        page_unlock(pp);
 799                        phm = PAGE_HASH_MUTEX(index);
 800                        mutex_enter(phm);
 801                        hash_locked = 1;
 802                        goto top;
 803                }

 805                /*
 806                 * If page_trylock() was called, then pp may still be on
 807                 * the cachelist (can’t be on the free list, it would not
 808                 * have been found in the search).  If it is on the
 809                 * cachelist it must be pulled now. To pull the page from
 810                 * the cachelist, it must be exclusively locked.
 811                 *
 812                 * The other big difference between page_trylock() and
 813                 * page_lock(), is that page_lock() will pull the
 814                 * page from whatever free list (the cache list in this
 815                 * case) the page is on.  If page_trylock() was used
 816                 * above, then we have to do the reclaim ourselves.
 817                 */
 818                if ((!hash_locked) && (PP_ISFREE(pp))) {
 819                        ASSERT(PP_ISAGED(pp) == 0);
```



```
 820                        VM_STAT_ADD(page_lookup_cnt[8]);

 822                        /*
 823                         * page_relcaim will insure that we
 824                         * have this page exclusively
 825                         */

 827                        if (!page_reclaim(pp, NULL)) {
 828                                /*
 829                                 * Page_reclaim dropped whatever lock
 830                                 * we held.
 831                                 */
 832                                VM_STAT_ADD(page_lookup_cnt[9]);
 833                                phm = PAGE_HASH_MUTEX(index);
 834                                mutex_enter(phm);
 835                                hash_locked = 1;
 836                                goto top;
 837                        } else if (se == SE_SHARED && newpp == NULL) {
 838                                VM_STAT_ADD(page_lookup_cnt[10]);
 839                                page_downgrade(pp);
 840                        }
 841                }

 843                if (hash_locked) {
 844                        mutex_exit(phm);
 845                }

 847                if (newpp != NULL && pp->p_szc < newpp->p_szc &&
 848                    PAGE_EXCL(pp) && nrelocp != NULL) {
 849                        ASSERT(nrelocp != NULL);
 850                        (void) page_relocate(&pp, &newpp, 1, 1, nrelocp,
 851                            NULL);
 852                        if (*nrelocp > 0) {
 853                                VM_STAT_COND_ADD(*nrelocp == 1,
 854                                    page_lookup_cnt[11]);
 855                                VM_STAT_COND_ADD(*nrelocp > 1,
 856                                    page_lookup_cnt[12]);
 857                                pp = newpp;
 858                                se = SE_EXCL;
 859                        } else {
 860                                if (se == SE_SHARED) {
 861                                        page_downgrade(pp);
 862                                }
 863                                VM_STAT_ADD(page_lookup_cnt[13]);
 864                        }
 865                } else if (newpp != NULL && nrelocp != NULL) {
 866                        if (PAGE_EXCL(pp) && se == SE_SHARED) {
 867                                page_downgrade(pp);
 868                        }
 869                        VM_STAT_COND_ADD(pp->p_szc < newpp->p_szc,
 870                            page_lookup_cnt[14]);
 871                        VM_STAT_COND_ADD(pp->p_szc == newpp->p_szc,
 872                            page_lookup_cnt[15]);
 873                        VM_STAT_COND_ADD(pp->p_szc > newpp->p_szc,
 874                            page_lookup_cnt[16]);
 875                } else if (newpp != NULL && PAGE_EXCL(pp)) {
 876                        se = SE_EXCL;
 877                }
 878        } else if (!hash_locked) {
 879                VM_STAT_ADD(page_lookup_cnt[17]);
 880                phm = PAGE_HASH_MUTEX(index);
 881                mutex_enter(phm);
 882                hash_locked = 1;
 883                goto top;
 884        } else if (newpp != NULL) {
 885                /*
```

```
 886                  * If we have a preallocated page then
 887                  * insert it now and basically behave like
 888                  * page_create.
 889                  */
 890                 VM_STAT_ADD(page_lookup_cnt[18]);
 891                 /*
 892                  * Since we hold the page hash mutex and
 893                  * just searched for this page, page_hashin
 894                  * had better not fail.  If it does, that
 895                  * means some thread did not follow the
 896                  * page hash mutex rules.  Panic now and
 897                  * get it over with.  As usual, go down
 898                  * holding all the locks.
 899                  */
 900                 ASSERT(MUTEX_HELD(phm));
 901                 if (!page_hashin(newpp, vp, off, phm)) {
 902                         ASSERT(MUTEX_HELD(phm));
 903                         panic("page_lookup_create: hashin failed %p %p %llx %p",
 904                             (void *)newpp, (void *)vp, off, (void *)phm);
 905                         /*NOTREACHED*/
 906                 }
 907                 ASSERT(MUTEX_HELD(phm));
 908                 mutex_exit(phm);
 909                 phm = NULL;
 910                 page_set_props(newpp, P_REF);
 911                 page_io_lock(newpp);
 912                 pp = newpp;
 913                 se = SE_EXCL;
 914         } else {
 915                 VM_STAT_ADD(page_lookup_cnt[19]);
 916                 mutex_exit(phm);
 917         }

 919         ASSERT(pp ? PAGE_LOCKED_SE(pp, se) : 1);

 921         ASSERT(pp ? ((PP_ISFREE(pp) == 0) && (PP_ISAGED(pp) == 0)) : 1);

 923         return (pp);
 924 }

 926 /*
 927  * Search the hash list for the page representing the
 928  * specified [vp, offset] and return it locked.  Skip
 929  * free pages and pages that cannot be locked as requested.
 930  * Used while attempting to kluster pages.
 931  */
 932 page_t *
 933 page_lookup_nowait(vnode_t *vp, u_offset_t off, se_t se)
 934 {
 935         page_t          *pp;
 936         kmutex_t        *phm;
 937         ulong_t         index;
 938         uint_t          locked;

 940         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
 941         VM_STAT_ADD(page_lookup_nowait_cnt[0]);

 943         index = PAGE_HASH_FUNC(vp, off);
 944         pp = page_hash_search(index, vp, off);
 254         PAGE_HASH_SEARCH(index, pp, vp, off);
 945         locked = 0;
 946         if (pp == NULL) {
 947 top:
 948                 VM_STAT_ADD(page_lookup_nowait_cnt[1]);
 949                 locked = 1;
 950                 phm = PAGE_HASH_MUTEX(index);
```



```
 951                 mutex_enter(phm);
 952                 pp = page_hash_search(index, vp, off);
 262                 PAGE_HASH_SEARCH(index, pp, vp, off);
 953         }

 955         if (pp == NULL || PP_ISFREE(pp)) {
 956                 VM_STAT_ADD(page_lookup_nowait_cnt[2]);
 957                 pp = NULL;
 958         } else {
 959                 if (!page_trylock(pp, se)) {
 960                         VM_STAT_ADD(page_lookup_nowait_cnt[3]);
 961                         pp = NULL;
 962                 } else {
 963                         VM_STAT_ADD(page_lookup_nowait_cnt[4]);
 964                         /*
 965                          * See the comment in page_lookup()
 966                          */
 967                         if (((volatile struct vnode *)(pp->p_vnode) != vp) ||
 968                             ((u_offset_t)(pp->p_offset) != off)) {
 969                                 VM_STAT_ADD(page_lookup_nowait_cnt[5]);
 970                                 if (locked) {
 971                                         panic("page_lookup_nowait %p",
 972                                             (void *)pp);
 973                                         /*NOTREACHED*/
 974                                 }
 975                                 page_unlock(pp);
 976                                 goto top;
 977                         }
 978                         if (PP_ISFREE(pp)) {
 979                                 VM_STAT_ADD(page_lookup_nowait_cnt[6]);
 980                                 page_unlock(pp);
 981                                 pp = NULL;
 982                         }
 983                 }
 984         }
 985         if (locked) {
 986                 VM_STAT_ADD(page_lookup_nowait_cnt[7]);
 987                 mutex_exit(phm);
 988         }

 990         ASSERT(pp ? PAGE_LOCKED_SE(pp, se) : 1);

 992         return (pp);
 993 }

 995 /*
 996  * Search the hash list for a page with the specified [vp, off]
 997  * that is known to exist and is already locked.  This routine
 998  * is typically used by segment SOFTUNLOCK routines.
 999  */
1000 page_t *
1001 page_find(vnode_t *vp, u_offset_t off)
1002 {
1003         page_t          *pp;
1004         kmutex_t        *phm;
1005         ulong_t         index;

1007         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
1008         VM_STAT_ADD(page_find_cnt);

1010         index = PAGE_HASH_FUNC(vp, off);
1011         phm = PAGE_HASH_MUTEX(index);

1013         mutex_enter(phm);
1014         pp = page_hash_search(index, vp, off);
 324         PAGE_HASH_SEARCH(index, pp, vp, off);
```

```
1015         mutex_exit(phm);

1017         ASSERT(pp == NULL || PAGE_LOCKED(pp) || panicstr);
1018         return (pp);
1019 }

1021 /*
1022  * Determine whether a page with the specified [vp, off]
1023  * currently exists in the system.  Obviously this should
1024  * only be considered as a hint since nothing prevents the
1025  * page from disappearing or appearing immediately after
1026  * the return from this routine. Subsequently, we don't
1027  * even bother to lock the list.
1028  */
1029 page_t *
1030 page_exists(vnode_t *vp, u_offset_t off)
1031 {
 342         page_t  *pp;
1032         ulong_t         index;

1034         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
1035         VM_STAT_ADD(page_exists_cnt);

1037         index = PAGE_HASH_FUNC(vp, off);
 349         PAGE_HASH_SEARCH(index, pp, vp, off);

1039         return (page_hash_search(index, vp, off));
 351         return (pp);
1040 }

1042 /*
1043  * Determine if physically contiguous pages exist for [vp, off] - [vp, off +
1044  * page_size(szc)) range.  if they exist and ppa is not NULL fill ppa array
1045  * with these pages locked SHARED. If necessary reclaim pages from
1046  * freelist. Return 1 if contiguous pages exist and 0 otherwise.
1047  *
1048  * If we fail to lock pages still return 1 if pages exist and contiguous.
1049  * But in this case return value is just a hint. ppa array won't be filled.
1050  * Caller should initialize ppa[0] as NULL to distinguish return value.
1051  *
1052  * Returns 0 if pages don't exist or not physically contiguous.
1053  *
1054  * This routine doesn't work for anonymous(swapfs) pages.
1055  */
1056 int
1057 page_exists_physcontig(vnode_t *vp, u_offset_t off, uint_t szc, page_t *ppa[])
1058 {
1059         pgcnt_t pages;
1060         pfn_t pfn;
1061         page_t *rootpp;
1062         pgcnt_t i;
1063         pgcnt_t j;
1064         u_offset_t save_off = off;
1065         ulong_t index;
1066         kmutex_t *phm;
1067         page_t *pp;
1068         uint_t pszc;
1069         int loopcnt = 0;

1071         ASSERT(szc != 0);
1072         ASSERT(vp != NULL);
1073         ASSERT(!IS_SWAPFSVP(vp));
1074         ASSERT(!VN_ISKAS(vp));

1076 again:
1077         if (++loopcnt > 3) {
```



```
1078                 VM_STAT_ADD(page_exphcontg[0]);
1079                 return (0);
1080         }

1082         index = PAGE_HASH_FUNC(vp, off);
1083         phm = PAGE_HASH_MUTEX(index);

1085         mutex_enter(phm);
1086         pp = page_hash_search(index, vp, off);
 398         PAGE_HASH_SEARCH(index, pp, vp, off);
1087         mutex_exit(phm);

1089         VM_STAT_ADD(page_exphcontg[1]);

1091         if (pp == NULL) {
1092                 VM_STAT_ADD(page_exphcontg[2]);
1093                 return (0);
1094         }

1096         pages = page_get_pagecnt(szc);
1097         rootpp = pp;
1098         pfn = rootpp->p_pagenum;

1100         if ((pszc = pp->p_szc) >= szc && ppa != NULL) {
1101                 VM_STAT_ADD(page_exphcontg[3]);
1102                 if (!page_trylock(pp, SE_SHARED)) {
1103                         VM_STAT_ADD(page_exphcontg[4]);
1104                         return (1);
1105                 }
1106                 /*
1107                  * Also check whether p_pagenum was modified by DR.
1108                  */
1109                 if (pp->p_szc != pszc || pp->p_vnode != vp ||
1110                     pp->p_offset != off || pp->p_pagenum != pfn) {
1111                         VM_STAT_ADD(page_exphcontg[5]);
1112                         page_unlock(pp);
1113                         off = save_off;
1114                         goto again;
1115                 }
1116                 /*
1117                  * szc was non zero and vnode and offset matched after we
1118                  * locked the page it means it can't become free on us.
1119                  */
1120                 ASSERT(!PP_ISFREE(pp));
1121                 if (!IS_P2ALIGNED(pfn, pages)) {
1122                         page_unlock(pp);
1123                         return (0);
1124                 }
1125                 ppa[0] = pp;
1126                 pp++;
1127                 off += PAGESIZE;
1128                 pfn++;
1129                 for (i = 1; i < pages; i++, pp++, off += PAGESIZE, pfn++) {
1130                         if (!page_trylock(pp, SE_SHARED)) {
1131                                 VM_STAT_ADD(page_exphcontg[6]);
1132                                 pp--;
1133                                 while (i-- > 0) {
1134                                         page_unlock(pp);
1135                                         pp--;
1136                                 }
1137                                 ppa[0] = NULL;
1138                                 return (1);
1139                         }
1140                         if (pp->p_szc != pszc) {
1141                                 VM_STAT_ADD(page_exphcontg[7]);
1142                                 page_unlock(pp);
```

```
1143                                    pp--;
1144                                    while (i-- > 0) {
1145                                            page_unlock(pp);
1146                                            pp--;
1147                                    }
1148                                    ppa[0] = NULL;
1149                                    off = save_off;
1150                                    goto again;
1151                            }
1152                            /*
1153                             * szc the same as for previous already locked pages
1154                             * with right identity. Since this page had correct
1155                             * szc after we locked it can't get freed or destroyed
1156                             * and therefore must have the expected identity.
1157                             */
1158                            ASSERT(!PP_ISFREE(pp));
1159                            if (pp->p_vnode != vp ||
1160                                pp->p_offset != off) {
1161                                panic("page_exists_physcontig: "
1162                                    "large page identity doesn't match");
1163                            }
1164                            ppa[i] = pp;
1165                            ASSERT(pp->p_pagenum == pfn);
1166                    }
1167                    VM_STAT_ADD(page_exphcontg[8]);
1168                    ppa[pages] = NULL;
1169                    return (1);
1170            } else if (pszc >= szc) {
1171                    VM_STAT_ADD(page_exphcontg[9]);
1172                    if (!IS_P2ALIGNED(pfn, pages)) {
1173                            return (0);
1174                    }
1175                    return (1);
1176            }

1178            if (!IS_P2ALIGNED(pfn, pages)) {
1179                    VM_STAT_ADD(page_exphcontg[10]);
1180                    return (0);
1181            }

1183            if (page_numtomemseg_nolock(pfn) !=
1184                page_numtomemseg_nolock(pfn + pages - 1)) {
1185                    VM_STAT_ADD(page_exphcontg[11]);
1186                    return (0);
1187            }

1189            /*
1190             * We loop up 4 times across pages to promote page size.
1191             * We're extra cautious to promote page size atomically with respect
1192             * to everybody else.  But we can probably optimize into 1 loop if
1193             * this becomes an issue.
1194             */

1196            for (i = 0; i < pages; i++, pp++, off += PAGESIZE, pfn++) {
1197                    if (!page_trylock(pp, SE_EXCL)) {
1198                            VM_STAT_ADD(page_exphcontg[12]);
1199                            break;
1200                    }
1201                    /*
1202                     * Check whether p_pagenum was modified by DR.
1203                     */
1204                    if (pp->p_pagenum != pfn) {
1205                            page_unlock(pp);
1206                            break;
1207                    }
1208                    if (pp->p_vnode != vp ||
```



```
1209                        pp->p_offset != off) {
1210                            VM_STAT_ADD(page_exphcontg[13]);
1211                            page_unlock(pp);
1212                            break;
1213                    }
1214                    if (pp->p_szc >= szc) {
1215                            ASSERT(i == 0);
1216                            page_unlock(pp);
1217                            off = save_off;
1218                            goto again;
1219                    }
1220            }

1222            if (i != pages) {
1223                    VM_STAT_ADD(page_exphcontg[14]);
1224                    --pp;
1225                    while (i-- > 0) {
1226                            page_unlock(pp);
1227                            --pp;
1228                    }
1229                    return (0);
1230            }

1232            pp = rootpp;
1233            for (i = 0; i < pages; i++, pp++) {
1234                    if (PP_ISFREE(pp)) {
1235                            VM_STAT_ADD(page_exphcontg[15]);
1236                            ASSERT(!PP_ISAGED(pp));
1237                            ASSERT(pp->p_szc == 0);
1238                            if (!page_reclaim(pp, NULL)) {
1239                                    break;
1240                            }
1241                    } else {
1242                            ASSERT(pp->p_szc < szc);
1243                            VM_STAT_ADD(page_exphcontg[16]);
1244                            (void) hat_pageunload(pp, HAT_FORCE_PGUNLOAD);
1245                    }
1246            }
1247            if (i < pages) {
1248                    VM_STAT_ADD(page_exphcontg[17]);
1249                    /*
1250                     * page_reclaim failed because we were out of memory.
1251                     * drop the rest of the locks and return because this page
1252                     * must be already reallocated anyway.
1253                     */
1254                    pp = rootpp;
1255                    for (j = 0; j < pages; j++, pp++) {
1256                            if (j != i) {
1257                                    page_unlock(pp);
1258                            }
1259                    }
1260                    return (0);
1261            }

1263            off = save_off;
1264            pp = rootpp;
1265            for (i = 0; i < pages; i++, pp++, off += PAGESIZE) {
1266                    ASSERT(PAGE_EXCL(pp));
1267                    ASSERT(!PP_ISFREE(pp));
1268                    ASSERT(!hat_page_is_mapped(pp));
1269                    ASSERT(pp->p_vnode == vp);
1270                    ASSERT(pp->p_offset == off);
1271                    pp->p_szc = szc;
1272            }
1273            pp = rootpp;
1274            for (i = 0; i < pages; i++, pp++) {
```

```
1275                 if (ppa == NULL) {
1276                         page_unlock(pp);
1277                 } else {
1278                         ppa[i] = pp;
1279                         page_downgrade(ppa[i]);
1280                 }
1281         }
1282         if (ppa != NULL) {
1283                 ppa[pages] = NULL;
1284         }
1285         VM_STAT_ADD(page_exphcontg[18]);
1286         ASSERT(vp->v_pages != NULL);
1287         return (1);
1288 }

1290 /*
1291  * Determine whether a page with the specified [vp, off]
1292  * currently exists in the system and if so return its
1293  * size code. Obviously this should only be considered as
1294  * a hint since nothing prevents the page from disappearing
1295  * or appearing immediately after the return from this routine.
1296  */
1297 int
1298 page_exists_forreal(vnode_t *vp, u_offset_t off, uint_t *szc)
1299 {
1300         page_t          *pp;
1301         kmutex_t        *phm;
1302         ulong_t         index;
1303         int             rc = 0;

1305         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
1306         ASSERT(szc != NULL);
1307         VM_STAT_ADD(page_exists_forreal_cnt);

1309         index = PAGE_HASH_FUNC(vp, off);
1310         phm = PAGE_HASH_MUTEX(index);

1312         mutex_enter(phm);
1313         pp = page_hash_search(index, vp, off);
 625         PAGE_HASH_SEARCH(index, pp, vp, off);
1314         if (pp != NULL) {
1315                 *szc = pp->p_szc;
1316                 rc = 1;
1317         }
1318         mutex_exit(phm);
1319         return (rc);
1320 }
```

_____***unchanged_portion_omitted_***

```
2253 page_t *
2254 page_create_va(vnode_t *vp, u_offset_t off, size_t bytes, uint_t flags,
2255     struct seg *seg, caddr_t vaddr)
2256 {
2257         page_t          *plist = NULL;
2258         pgcnt_t         npages;
2259         pgcnt_t         found_on_free = 0;
2260         pgcnt_t         pages_req;
2261         page_t          *npp = NULL;
2262         struct pcf      *p;
2263         lgrp_t          *lgrp;

2265         TRACE_4(TR_FAC_VM, TR_PAGE_CREATE_START,
2266             "page_create_start:vp %p off %llx bytes %lu flags %x",
2267             vp, off, bytes, flags);

2269         ASSERT(bytes != 0 && vp != NULL);
```



```
2271         if ((flags & PG_EXCL) == 0 && (flags & PG_WAIT) == 0) {
2272                 panic("page_create: invalid flags");
2273                 /*NOTREACHED*/
2274         }
2275         ASSERT((flags & ~(PG_EXCL | PG_WAIT |
2276             PG_NORELOC | PG_PANIC | PG_PUSHPAGE | PG_NORMALPRI)) == 0);
2277             /* but no others */

2279         pages_req = npages = btopr(bytes);
2280         /*
2281          * Try to see whether request is too large to *ever* be
2282          * satisfied, in order to prevent deadlock.  We arbitrarily
2283          * decide to limit maximum size requests to max_page_get.
2284          */
2285         if (npages >= max_page_get) {
2286                 if ((flags & PG_WAIT) == 0) {
2287                         TRACE_4(TR_FAC_VM, TR_PAGE_CREATE_TOOBIG,
2288                             "page_create_toobig:vp %p off %llx npages "
2289                             "%lu max_page_get %lu",
2290                             vp, off, npages, max_page_get);
2291                         return (NULL);
2292                 } else {
2293                         cmn_err(CE_WARN,
2294                             "Request for too much kernel memory "
2295                             "(%lu bytes), will hang forever", bytes);
2296                         for (;;)
2297                                 delay(1000000000);
2298                 }
2299         }

2301         if (!kcage_on || panicstr) {
2302                 /*
2303                  * Cage is OFF, or we are single threaded in
2304                  * panic, so make everything a RELOC request.
2305                  */
2306                 flags &= ~PG_NORELOC;
2307         }

2309         if (freemem <= throttlefree + npages)
2310                 if (!page_create_throttle(npages, flags))
2311                         return (NULL);

2313         /*
2314          * If cage is on, dampen draw from cage when available
2315          * cage space is low.
2316          */
2317         if ((flags & PG_NORELOC) &&
2318             kcage_freemem < kcage_throttlefree + npages) {

2320                 /*
2321                  * The cage is on, the caller wants PG_NORELOC
2322                  * pages and available cage memory is very low.
2323                  * Call kcage_create_throttle() to attempt to
2324                  * control demand on the cage.
2325                  */
2326                 if (kcage_create_throttle(npages, flags) == KCT_FAILURE)
2327                         return (NULL);
2328         }

2330         VM_STAT_ADD(page_create_cnt[0]);

2332         if (!pcf_decrement_bucket(npages)) {
2333                 /*
2334                  * Have to look harder.  If npages is greater than
2335                  * one, then we might have to coalesce the counters.
```

```
2336                     *
2337                     * Go wait.  We come back having accounted
2338                     * for the memory.
2339                     */
2340                    VM_STAT_ADD(page_create_cnt[1]);
2341                    if (!page_create_wait(npages, flags)) {
2342                            VM_STAT_ADD(page_create_cnt[2]);
2343                            return (NULL);
2344                    }
2345            }

2347            TRACE_2(TR_FAC_VM, TR_PAGE_CREATE_SUCCESS,
2348                "page_create_success:vp %p off %llx", vp, off);

2350            /*
2351             * If satisfying this request has left us with too little
2352             * memory, start the wheels turning to get some back.  The
2353             * first clause of the test prevents waking up the pageout
2354             * daemon in situations where it would decide that there's
2355             * nothing to do.
2356             */
2357            if (nscan < desscan && freemem < minfree) {
2358                    TRACE_1(TR_FAC_VM, TR_PAGEOUT_CV_SIGNAL,
2359                        "pageout_cv_signal:freemem %ld", freemem);
2360                    cv_signal(&proc_pageout->p_cv);
2361            }

2363            /*
2364             * Loop around collecting the requested number of pages.
2365             * Most of the time, we have to `create' a new page. With
2366             * this in mind, pull the page off the free list before
2367             * getting the hash lock.  This will minimize the hash
2368             * lock hold time, nesting, and the like.  If it turns
2369             * out we don't need the page, we put it back at the end.
2370             */
2371            while (npages--) {
2372                    page_t          *pp;
2373                    kmutex_t        *phm = NULL;
2374                    ulong_t         index;

2376                    index = PAGE_HASH_FUNC(vp, off);
2377 top:
2378                    ASSERT(phm == NULL);
2379                    ASSERT(index == PAGE_HASH_FUNC(vp, off));
2380                    ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));

2382                    if (npp == NULL) {
2383                            /*
2384                             * Try to get a page from the freelist (ie,
2385                             * a page with no [vp, off] tag).  If that
2386                             * fails, use the cachelist.
2387                             *
2388                             * During the first attempt at both the free
2389                             * and cache lists we try for the correct color.
2390                             */
2391                            /*
2392                             * XXXX-how do we deal with virtual indexed
2393                             * caches and and colors?
2394                             */
2395                            VM_STAT_ADD(page_create_cnt[4]);
2396                            /*
2397                             * Get lgroup to allocate next page of shared memory
2398                             * from and use it to specify where to allocate
2399                             * the physical memory
2400                             */
2401                            lgrp = lgrp_mem_choose(seg, vaddr, PAGESIZE);
```



```
2402                            npp = page_get_freelist(vp, off, seg, vaddr, PAGESIZE,
2403                                flags | PG_MATCH_COLOR, lgrp);
2404                            if (npp == NULL) {
2405                                    npp = page_get_cachelist(vp, off, seg,
2406                                        vaddr, flags | PG_MATCH_COLOR, lgrp);
2407                                    if (npp == NULL) {
2408                                            npp = page_create_get_something(vp,
2409                                                off, seg, vaddr,
2410                                                flags & ~PG_MATCH_COLOR);
2411                                    }

2413                                    if (PP_ISAGED(npp) == 0) {
2414                                            /*
2415                                             * Since this page came from the
2416                                             * cachelist, we must destroy the
2417                                             * old vnode association.
2418                                             */
2419                                            page_hashout(npp, NULL);
2420                                    }
2421                            }
2422                    }

2424                    /*
2425                     * We own this page!
2426                     */
2427                    ASSERT(PAGE_EXCL(npp));
2428                    ASSERT(npp->p_vnode == NULL);
2429                    ASSERT(!hat_page_is_mapped(npp));
2430                    PP_CLRFREE(npp);
2431                    PP_CLRAGED(npp);

2433                    /*
2434                     * Here we have a page in our hot little mits and are
2435                     * just waiting to stuff it on the appropriate lists.
2436                     * Get the mutex and check to see if it really does
2437                     * not exist.
2438                     */
2439                    phm = PAGE_HASH_MUTEX(index);
2440                    mutex_enter(phm);
2441                    pp = page_hash_search(index, vp, off);
1753                    PAGE_HASH_SEARCH(index, pp, vp, off);
2442                    if (pp == NULL) {
2443                            VM_STAT_ADD(page_create_new);
2444                            pp = npp;
2445                            npp = NULL;
2446                            if (!page_hashin(pp, vp, off, phm)) {
2447                                    /*
2448                                     * Since we hold the page hash mutex and
2449                                     * just searched for this page, page_hashin
2450                                     * had better not fail.  If it does, that
2451                                     * means somethread did not follow the
2452                                     * page hash mutex rules.  Panic now and
2453                                     * get it over with.  As usual, go down
2454                                     * holding all the locks.
2455                                     */
2456                                    ASSERT(MUTEX_HELD(phm));
2457                                    panic("page_create: "
2458                                        "hashin failed %p %p %llx %p",
2459                                        (void *)pp, (void *)vp, off, (void *)phm);
2460                                    /*NOTREACHED*/
2461                            }
2462                            ASSERT(MUTEX_HELD(phm));
2463                            mutex_exit(phm);
2464                            phm = NULL;

2466                            /*
```

```
2467                         * Hat layer locking need not be done to set
2468                         * the following bits since the page is not hashed
2469                         * and was on the free list (i.e., had no mappings).
2470                         *
2471                         * Set the reference bit to protect
2472                         * against immediate pageout
2473                         *
2474                         * XXXmh modify freelist code to set reference
2475                         * bit so we don't have to do it here.
2476                         */
2477                        page_set_props(pp, P_REF);
2478                        found_on_free++;
2479                } else {
2480                        VM_STAT_ADD(page_create_exists);
2481                        if (flags & PG_EXCL) {
2482                                /*
2483                                 * Found an existing page, and the caller
2484                                 * wanted all new pages.  Undo all of the work
2485                                 * we have done.
2486                                 */
2487                                mutex_exit(phm);
2488                                phm = NULL;
2489                                while (plist != NULL) {
2490                                        pp = plist;
2491                                        page_sub(&plist, pp);
2492                                        page_io_unlock(pp);
2493                                        /* large pages should not end up here */
2494                                        ASSERT(pp->p_szc == 0);
2495                                        /*LINTED: constant in conditional ctx*/
2496                                        VN_DISPOSE(pp, B_INVAL, 0, kcred);
2497                                }
2498                                VM_STAT_ADD(page_create_found_one);
2499                                goto fail;
2500                        }
2501                        ASSERT(flags & PG_WAIT);
2502                        if (!page_lock(pp, SE_EXCL, phm, P_NO_RECLAIM)) {
2503                                /*
2504                                 * Start all over again if we blocked trying
2505                                 * to lock the page.
2506                                 */
2507                                mutex_exit(phm);
2508                                VM_STAT_ADD(page_create_page_lock_failed);
2509                                phm = NULL;
2510                                goto top;
2511                        }
2512                        mutex_exit(phm);
2513                        phm = NULL;

2515                        if (PP_ISFREE(pp)) {
2516                                ASSERT(PP_ISAGED(pp) == 0);
2517                                VM_STAT_ADD(pagecnt.pc_get_cache);
2518                                page_list_sub(pp, PG_CACHE_LIST);
2519                                PP_CLRFREE(pp);
2520                                found_on_free++;
2521                        }
2522                }

2524                /*
2525                 * Got a page!  It is locked.  Acquire the i/o
2526                 * lock since we are going to use the p_next and
2527                 * p_prev fields to link the requested pages together.
2528                 */
2529                page_io_lock(pp);
2530                page_add(&plist, pp);
2531                plist = plist->p_next;
2532                off += PAGESIZE;
```



```
2533                vaddr += PAGESIZE;
2534        }

2536        ASSERT((flags & PG_EXCL) ? (found_on_free == pages_req) : 1);
2537 fail:
2538        if (npp != NULL) {
2539                /*
2540                 * Did not need this page after all.
2541                 * Put it back on the free list.
2542                 */
2543                VM_STAT_ADD(page_create_putbacks);
2544                PP_SETFREE(npp);
2545                PP_SETAGED(npp);
2546                npp->p_offset = (u_offset_t)-1;
2547                page_list_add(npp, PG_FREE_LIST | PG_LIST_TAIL);
2548                page_unlock(npp);

2550        }

2552        ASSERT(pages_req >= found_on_free);

2554        {
2555                uint_t overshoot = (uint_t)(pages_req - found_on_free);

2557                if (overshoot) {
2558                        VM_STAT_ADD(page_create_overshoot);
2559                        p = &pcf[PCF_INDEX()];
2560                        mutex_enter(&p->pcf_lock);
2561                        if (p->pcf_block) {
2562                                p->pcf_reserve += overshoot;
2563                        } else {
2564                                p->pcf_count += overshoot;
2565                                if (p->pcf_wait) {
2566                                        mutex_enter(&new_freemem_lock);
2567                                        if (freemem_wait) {
2568                                                cv_signal(&freemem_cv);
2569                                                p->pcf_wait--;
2570                                        } else {
2571                                                p->pcf_wait = 0;
2572                                        }
2573                                        mutex_exit(&new_freemem_lock);
2574                                }
2575                        }
2576                        mutex_exit(&p->pcf_lock);
2577                        /* freemem is approximate, so this test OK */
2578                        if (!p->pcf_block)
2579                                freemem += overshoot;
2580                }
2581        }

2583        return (plist);
2584 }
_____unchanged_portion_omitted_

3204 /*
3205  * Rename the page "opp" to have an identity specified
3206  * by [vp, off].  If a page already exists with this name
3207  * it is locked and destroyed.  Note that the page's
3208  * translations are not unloaded during the rename.
3209  *
3210  * This routine is used by the anon layer to "steal" the
3211  * original page and is not unlike destroying a page and
3212  * creating a new page using the same page frame.
3213  *
3214  * XXX -- Could deadlock if caller 1 tries to rename A to B while
3215  * caller 2 tries to rename B to A.
```

```
3216  */
3217 void
3218 page_rename(page_t *opp, vnode_t *vp, u_offset_t off)
3219 {
3220         page_t          *pp;
3221         int             olckcnt = 0;
3222         int             ocowcnt = 0;
3223         kmutex_t        *phm;
3224         ulong_t         index;

3226         ASSERT(PAGE_EXCL(opp) && !page_iolock_assert(opp));
3227         ASSERT(MUTEX_NOT_HELD(page_vnode_mutex(vp)));
3228         ASSERT(PP_ISFREE(opp) == 0);

3230         VM_STAT_ADD(page_rename_count);

3232         TRACE_3(TR_FAC_VM, TR_PAGE_RENAME,
3233             "page rename:pp %p vp %p off %llx", opp, vp, off);

3235         /*
3236          * CacheFS may call page_rename for a large NFS page
3237          * when both CacheFS and NFS mount points are used
3238          * by applications. Demote this large page before
3239          * renaming it, to ensure that there are no "partial"
3240          * large pages left lying around.
3241          */
3242         if (opp->p_szc != 0) {
3243                 vnode_t *ovp = opp->p_vnode;
3244                 ASSERT(ovp != NULL);
3245                 ASSERT(!IS_SWAPFSVP(ovp));
3246                 ASSERT(!VN_ISKAS(ovp));
3247                 page_demote_vp_pages(opp);
3248                 ASSERT(opp->p_szc == 0);
3249         }

3251         page_hashout(opp, NULL);
3252         PP_CLRAGED(opp);

3254         /*
3255          * Acquire the appropriate page hash lock, since
3256          * we're going to rename the page.
3257          */
3258         index = PAGE_HASH_FUNC(vp, off);
3259         phm = PAGE_HASH_MUTEX(index);
3260         mutex_enter(phm);
3261 top:
3262         /*
3263          * Look for an existing page with this name and destroy it if found.
3264          * By holding the page hash lock all the way to the page_hashin()
3265          * call, we are assured that no page can be created with this
3266          * identity.  In the case when the phm lock is dropped to undo any
3267          * hat layer mappings, the existing page is held with an "exclusive"
3268          * lock, again preventing another page from being created with
3269          * this identity.
3270          */
3271         pp = page_hash_search(index, vp, off);
2583         PAGE_HASH_SEARCH(index, pp, vp, off);
3272         if (pp != NULL) {
3273                 VM_STAT_ADD(page_rename_exists);

3275                 /*
3276                  * As it turns out, this is one of only two places where
3277                  * page_lock() needs to hold the passed in lock in the
3278                  * successful case.  In all of the others, the lock could
3279                  * be dropped as soon as the attempt is made to lock
3280                  * the page.  It is tempting to add yet another arguement,
```



```
3281                  * PL_KEEP or PL_DROP, to let page_lock know what to do.
3282                  */
3283                 if (!page_lock(pp, SE_EXCL, phm, P_RECLAIM)) {
3284                         /*
3285                          * Went to sleep because the page could not
3286                          * be locked.  We were woken up when the page
3287                          * was unlocked, or when the page was destroyed.
3288                          * In either case, 'phm' was dropped while we
3289                          * slept.  Hence we should not just roar through
3290                          * this loop.
3291                          */
3292                         goto top;
3293                 }

3295                 /*
3296                  * If an existing page is a large page, then demote
3297                  * it to ensure that no "partial" large pages are
3298                  * "created" after page_rename. An existing page
3299                  * can be a CacheFS page, and can't belong to swapfs.
3300                  */
3301                 if (hat_page_is_mapped(pp)) {
3302                         /*
3303                          * Unload translations.  Since we hold the
3304                          * exclusive lock on this page, the page
3305                          * can not be changed while we drop phm.
3306                          * This is also not a lock protocol violation,
3307                          * but rather the proper way to do things.
3308                          */
3309                         mutex_exit(phm);
3310                         (void) hat_pageunload(pp, HAT_FORCE_PGUNLOAD);
3311                         if (pp->p_szc != 0) {
3312                                 ASSERT(!IS_SWAPFSVP(vp));
3313                                 ASSERT(!VN_ISKAS(vp));
3314                                 page_demote_vp_pages(pp);
3315                                 ASSERT(pp->p_szc == 0);
3316                         }
3317                         mutex_enter(phm);
3318                 } else if (pp->p_szc != 0) {
3319                         ASSERT(!IS_SWAPFSVP(vp));
3320                         ASSERT(!VN_ISKAS(vp));
3321                         mutex_exit(phm);
3322                         page_demote_vp_pages(pp);
3323                         ASSERT(pp->p_szc == 0);
3324                         mutex_enter(phm);
3325                 }
3326                 page_hashout(pp, phm);
3327         }
3328         /*
3329          * Hash in the page with the new identity.
3330          */
3331         if (!page_hashin(opp, vp, off, phm)) {
3332                 /*
3333                  * We were holding phm while we searched for [vp, off]
3334                  * and only dropped phm if we found and locked a page.
3335                  * If we can't create this page now, then some thing
3336                  * is really broken.
3337                  */
3338                 panic("page_rename: Can't hash in page: %p", (void *)pp);
3339                 /*NOTREACHED*/
3340         }

3342         ASSERT(MUTEX_HELD(phm));
3343         mutex_exit(phm);

3345         /*
3346          * Now that we have dropped phm, lets get around to finishing up
```

```
3347          * with pp.
3348          */
3349         if (pp != NULL) {
3350                 ASSERT(!hat_page_is_mapped(pp));
3351                 /* for now large pages should not end up here */
3352                 ASSERT(pp->p_szc == 0);
3353                 /*
3354                  * Save the locks for transfer to the new page and then
3355                  * clear them so page_free doesn't think they're important.
3356                  * The page_struct_lock need not be acquired for lckcnt and
3357                  * cowcnt since the page has an "exclusive" lock.
3358                  */
3359                 olckcnt = pp->p_lckcnt;
3360                 ocowcnt = pp->p_cowcnt;
3361                 pp->p_lckcnt = pp->p_cowcnt = 0;

3363                 /*
3364                  * Put the page on the "free" list after we drop
3365                  * the lock.  The less work under the lock the better.
3366                  */
3367                 /*LINTED: constant in conditional context*/
3368                 VN_DISPOSE(pp, B_FREE, 0, kcred);
3369         }

3371         /*
3372          * Transfer the lock count from the old page (if any).
3373          * The page_struct_lock need not be acquired for lckcnt and
3374          * cowcnt since the page has an "exclusive" lock.
3375          */
3376         opp->p_lckcnt += olckcnt;
3377         opp->p_cowcnt += ocowcnt;
3378 }
```

_____***unchanged_portion_omitted_***