```
**********************************************************
  280113 Fri May  8 18:05:13 2015
new/usr/src/uts/common/vm/seg_vn.c
PVN_GETPAGE_{SZ,NUM} are misnamed and unnecessarily complicated
There is really no reason to not allow 8 pages all the time.  With the
current logic, we get the following:
Assuming 4kB pages (x86):
    _SZ  = ptob(8) /* 32kB */
    _NUM = 8
Assuming 8kB pages (sparc):
    _SZ  = ptob(8) /* 64kB */
    _NUM = 8
We'd have to deal with 16kB base pages in order for the _NUM #define to not
be 8 (it'd be 4 in that case).  So, in the spirit of simplicity, let's just
always grab 8 pages as there are no interesting systems with 16kB+ base pages.
Finally, the defines are poorly named.
**********************************************************
```

```
/*
 * 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 (FAULT_TMP_PAGES_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.
 * 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 FAULT_TMP_PAGES_NUM     0x8
#define FAULT_TMP_PAGES_SZ      ptob(FAULT_TMP_PAGES_NUM)
#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 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_inherit(struct seg *, caddr_t, size_t, uint_t);

const struct seg_ops segvn_ops = {
        .dup            = segvn_dup,
        .unmap          = segvn_unmap,
        .free           = segvn_free,
        .fault          = segvn_fault,
        .faulta         = segvn_faulta,
        .setprot        = segvn_setprot,
        .checkprot      = segvn_checkprot,
        .kluster        = segvn_kluster,
        .sync           = segvn_sync,
        .incore         = segvn_incore,
        .lockop         = segvn_lockop,
        .getprot        = segvn_getprot,
        .getoffset      = segvn_getoffset,
        .gettype        = segvn_gettype,
        .getvp          = segvn_getvp,
        .advise         = segvn_advise,
        .dump           = segvn_dump,
        .pagelock       = segvn_pagelock,
        .setpagesize    = segvn_setpagesize,
        .getmemid       = segvn_getmemid,
        .getpolicy      = segvn_getpolicy,
        .inherit        = segvn_inherit,
};
```

______unchanged_portion_omitted_

```
int fltadvice = 1;      /* set to free behind pages for sequential access */

/*
 * This routine is called via a machine specific fault handling routine.
 * It is also called by software routines wishing to lock or unlock
 * a range of addresses.
 *
 * Here is the basic algorithm:
 *      If unlocking
 *              Call segvn_softunlock
 *              Return
 *      endif
 *      Checking and set up work
```

```
 *      If we will need some non-anonymous pages
 *              Call VOP_GETPAGE over the range of non-anonymous pages
 *      endif
 *      Loop over all addresses requested
 *              Call segvn_faultpage passing in page list
 *                  to load up translations and handle anonymous pages
 *      endloop
 *      Load up translation to any additional pages in page list not
 *          already handled that fit into this segment
 */
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)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        page_t **plp, **ppp, *pp;
        u_offset_t off;
        caddr_t a;
        struct vpage *vpage;
        uint_t vpprot, prot;
        int err;
        page_t *pl[FAULT_TMP_PAGES_NUM + 1];
        page_t *pl[PVN_GETPAGE_NUM + 1];
        size_t plsz, pl_alloc_sz;
        size_t page;
        ulong_t anon_index;
        struct anon_map *amp;
        int dogetpage = 0;
        caddr_t lpgaddr, lpgeaddr;
        size_t pgsz;
        anon_sync_obj_t cookie;
        int brkcow = BREAK_COW_SHARE(rw, type, svd->type);

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);

        /*
         * First handle the easy stuff
         */
        if (type == F_SOFTUNLOCK) {
                if (rw == S_READ_NOCOW) {
                        rw = S_READ;
                        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
                }
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
                pgsz = (seg->s_szc == 0) ? PAGESIZE :
                    page_get_pagesize(seg->s_szc);
                VM_STAT_COND_ADD(pgsz > PAGESIZE, segvnvmstats.fltanpages[16]);
                CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
                segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);
        }

        ASSERT(svd->tr_state == SEGVN_TR_OFF ||
            !HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
        if (brkcow == 0) {
                if (svd->tr_state == SEGVN_TR_INIT) {
                        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
                        if (svd->tr_state == SEGVN_TR_INIT) {
                                ASSERT(svd->vp != NULL && svd->amp == NULL);
                                ASSERT(svd->flags & MAP_TEXT);
                                ASSERT(svd->type == MAP_PRIVATE);
                                segvn_textrepl(seg);
                                ASSERT(svd->tr_state != SEGVN_TR_INIT);
                                ASSERT(svd->tr_state != SEGVN_TR_ON ||
                                    svd->amp != NULL);
                        }
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                }
        } else if (svd->tr_state != SEGVN_TR_OFF) {
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

                if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
                        ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (FC_PROT);
                }

                if (svd->tr_state == SEGVN_TR_ON) {
                        ASSERT(svd->vp != NULL && svd->amp != NULL);
                        segvn_textunrepl(seg, 0);
                        ASSERT(svd->amp == NULL &&
                            svd->tr_state == SEGVN_TR_OFF);
                } else if (svd->tr_state != SEGVN_TR_OFF) {
                        svd->tr_state = SEGVN_TR_OFF;
                }
                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        }

top:
        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

        /*
         * If we have the same protections for the entire segment,
         * insure that the access being attempted is legitimate.
         */

        if (svd->pageprot == 0) {
                uint_t protchk;

                switch (rw) {
                case S_READ:
                case S_READ_NOCOW:
                        protchk = PROT_READ;
                        break;
                case S_WRITE:
                        protchk = PROT_WRITE;
                        break;
                case S_EXEC:
                        protchk = PROT_EXEC;
                        break;
                case S_OTHER:
                default:
                        protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
                        break;
                }

                if ((svd->prot & protchk) == 0) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (FC_PROT);       /* illegal access type */
                }
        }

        if (brkcow && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                /* this must be SOFTLOCK S_READ fault */
                ASSERT(svd->amp == NULL);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
                if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
```

```
                        /*
                         * this must be the first ever non S_READ_NOCOW
                         * softlock for this segment.
                         */
                        ASSERT(svd->softlockcnt == 0);
                        hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                            HAT_REGION_TEXT);
                        svd->rcookie = HAT_INVALID_REGION_COOKIE;
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                goto top;
        }

        /*
         * We can't allow the long term use of softlocks for vmpss segments,
         * because in some file truncation cases we should be able to demote
         * the segment, which requires that there are no softlocks.  The
         * only case where it's ok to allow a SOFTLOCK fault against a vmpss
         * segment is S_READ_NOCOW, where the caller holds the address space
         * locked as writer and calls softunlock before dropping the as lock.
         * S_READ_NOCOW is used by /proc to read memory from another user.
         *
         * Another deadlock between SOFTLOCK and file truncation can happen
         * because segvn_fault_vnodepages() calls the FS one pagesize at
         * a time. A second VOP_GETPAGE() call by segvn_fault_vnodepages()
         * can cause a deadlock because the first set of page_t's remain
         * locked SE_SHARED.  To avoid this, we demote segments on a first
         * SOFTLOCK if they have a length greater than the segment's
         * page size.
         *
         * So for now, we only avoid demoting a segment on a SOFTLOCK when
         * the access type is S_READ_NOCOW and the fault length is less than
         * or equal to the segment's page size. While this is quite restrictive,
         * it should be the most common case of SOFTLOCK against a vmpss
         * segment.
         *
         * For S_READ_NOCOW, it's safe not to do a copy on write because the
         * caller makes sure no COW will be caused by another thread for a
         * softlocked page.
         */
        if (type == F_SOFTLOCK && svd->vp != NULL && seg->s_szc != 0) {
                int demote = 0;

                if (rw != S_READ_NOCOW) {
                        demote = 1;
                }
                if (!demote && len > PAGESIZE) {
                        pgsz = page_get_pagesize(seg->s_szc);
                        CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
                            lpgeaddr);
                        if (lpgeaddr - lpgaddr > pgsz) {
                                demote = 1;
                        }
                }

                ASSERT(demote || AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

                if (demote) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
                        if (seg->s_szc != 0) {
                                segvn_vmpss_clrszc_cnt++;
                                ASSERT(svd->softlockcnt == 0);
                                err = segvn_clrszc(seg);
                                if (err) {
                                        segvn_vmpss_clrszc_err++;
```

```c
                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                        return (FC_MAKE_ERR(err));
                                }
                        }
                        ASSERT(seg->s_szc == 0);
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        goto top;
                }
        }

        /*
         * Check to see if we need to allocate an anon_map structure.
         */
        if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                /*
                 * Drop the "read" lock on the segment and acquire
                 * the "write" version since we have to allocate the
                 * anon_map.
                 */
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

                if (svd->amp == NULL) {
                        svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
                        svd->amp->a_szc = seg->s_szc;
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

                /*
                 * Start all over again since segment protections
                 * may have changed after we dropped the "read" lock.
                 */
                goto top;
        }

        /*
         * S_READ_NOCOW vs S_READ distinction was
         * only needed for the code above. After
         * that we treat it as S_READ.
         */
        if (rw == S_READ_NOCOW) {
                ASSERT(type == F_SOFTLOCK);
                ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
                rw = S_READ;
        }

        amp = svd->amp;

        /*
         * MADV_SEQUENTIAL work is ignored for large page segments.
         */
        if (seg->s_szc != 0) {
                pgsz = page_get_pagesize(seg->s_szc);
                ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
                CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
                if (svd->vp == NULL) {
                        err = segvn_fault_anonpages(hat, seg, lpgaddr,
                            lpgeaddr, type, rw, addr, addr + len, brkcow);
                } else {
                        err = segvn_fault_vnodepages(hat, seg, lpgaddr,
                            lpgeaddr, type, rw, addr, addr + len, brkcow);
                        if (err == IE_RETRY) {
                                ASSERT(seg->s_szc == 0);
                                ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
```

```c
                                goto top;
                        }
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (err);
        }

        page = seg_page(seg, addr);
        if (amp != NULL) {
                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                anon_index = svd->anon_index + page;

                if (type == F_PROT && rw == S_READ &&
                    svd->tr_state == SEGVN_TR_OFF &&
                    svd->type == MAP_PRIVATE && svd->pageprot == 0) {
                        size_t index = anon_index;
                        struct anon *ap;

                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        /*
                         * The fast path could apply to S_WRITE also, except
                         * that the protection fault could be caused by lazy
                         * tlb flush when ro->rw. In this case, the pte is
                         * RW already. But RO in the other cpu's tlb causes
                         * the fault. Since hat_chgprot won't do anything if
                         * pte doesn't change, we may end up faulting
                         * indefinitely until the RO tlb entry gets replaced.
                         */
                        for (a = addr; a < addr + len; a += PAGESIZE, index++) {
                                anon_array_enter(amp, index, &cookie);
                                ap = anon_get_ptr(amp->ahp, index);
                                anon_array_exit(&cookie);
                                if ((ap == NULL) || (ap->an_refcnt != 1)) {
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                        goto slow;
                                }
                        }
                        hat_chgprot(seg->s_as->a_hat, addr, len, svd->prot);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
        }
slow:

        if (svd->vpage == NULL)
                vpage = NULL;
        else
                vpage = &svd->vpage[page];

        off = svd->offset + (uintptr_t)(addr - seg->s_base);

        /*
         * If MADV_SEQUENTIAL has been set for the particular page we
         * are faulting on, free behind all pages in the segment and put
         * them on the free list.
         */

        if ((page != 0) && fltadvice && svd->tr_state != SEGVN_TR_ON) {
                struct vpage *vpp;
                ulong_t fanon_index;
                size_t fpage;
                u_offset_t pgoff, fpgoff;
                struct vnode *fvp;
                struct anon *fap = NULL;
                if (svd->advice == MADV_SEQUENTIAL ||
                    (svd->pageadvice &&
                    VPP_ADVICE(vpage) == MADV_SEQUENTIAL)) {
                        pgoff = off - PAGESIZE;
                        fpage = page - 1;
                        if (vpage != NULL)
                                vpp = &svd->vpage[fpage];
                        if (amp != NULL)
                                fanon_index = svd->anon_index + fpage;

                        while (pgoff > svd->offset) {
                                if (svd->advice != MADV_SEQUENTIAL &&
                                    (!svd->pageadvice || (vpage &&
                                    VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
                                        break;

                                /*
                                 * If this is an anon page, we must find the
                                 * correct <vp, offset> for it
                                 */
                                fap = NULL;
                                if (amp != NULL) {
                                        ANON_LOCK_ENTER(&amp->a_rwlock,
                                            RW_READER);
                                        anon_array_enter(amp, fanon_index,
                                            &cookie);
                                        fap = anon_get_ptr(amp->ahp,
                                            fanon_index);
                                        if (fap != NULL) {
                                                swap_xlate(fap, &fvp, &fpgoff);
                                        } else {
                                                fpgoff = pgoff;
                                                fvp = svd->vp;
                                        }
                                        anon_array_exit(&cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                } else {
                                        fpgoff = pgoff;
                                        fvp = svd->vp;
                                }
                                if (fvp == NULL)
                                        break;  /* XXX */
                                /*
                                 * Skip pages that are free or have an
                                 * "exclusive" lock.
                                 */
                                pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
                                if (pp == NULL)
                                        break;
                                /*
                                 * We don’t need the page_struct_lock to test
                                 * as this is only advisory; even if we
                                 * acquire it someone might race in and lock
                                 * the page after we unlock and before the
                                 * PUTPAGE, then VOP_PUTPAGE will do nothing.
                                 */
                                if (pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
                                        /*
                                         * Hold the vnode before releasing
                                         * the page lock to prevent it from
                                         * being freed and re-used by some
                                         * other thread.
                                         */
                                        VN_HOLD(fvp);
                                        page_unlock(pp);
                                        /*
```

```
                                         * We should build a page list
                                         * to kluster putpages XXX
                                         */
                                        (void) VOP_PUTPAGE(fvp,
                                            (offset_t)fpgoff, PAGESIZE,
                                            (B_DONTNEED|B_FREE|B_ASYNC),
                                            svd->cred, NULL);
                                        VN_RELE(fvp);
                                } else {
                                        /*
                                         * XXX - Should the loop terminate if
                                         * the page is ‘locked’?
                                         */
                                        page_unlock(pp);
                                }
                                --vpp;
                                --fanon_index;
                                pgoff -= PAGESIZE;
                        }
                }
        }

        plp = pl;
        *plp = NULL;
        pl_alloc_sz = 0;

        /*
         * See if we need to call VOP_GETPAGE for
         * *any* of the range being faulted on.
         * We can skip all of this work if there
         * was no original vnode.
         */
        if (svd->vp != NULL) {
                u_offset_t vp_off;
                size_t vp_len;
                struct anon *ap;
                vnode_t *vp;

                vp_off = off;
                vp_len = len;

                if (amp == NULL)
                        dogetpage = 1;
                else {
                        /*
                         * Only acquire reader lock to prevent amp->ahp
                         * from being changed.  It’s ok to miss pages,
                         * hence we don’t do anon_array_enter
                         */
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        ap = anon_get_ptr(amp->ahp, anon_index);

                        if (len <= PAGESIZE)
                                /* inline non_anon() */
                                dogetpage = (ap == NULL);
                        else
                                dogetpage = non_anon(amp->ahp, anon_index,
                                    &vp_off, &vp_len);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }

                if (dogetpage) {
                        enum seg_rw arw;
                        struct as *as = seg->s_as;

                        if (len > FAULT_TMP_PAGES_SZ) {
```

```
                        if (len > ptob((sizeof (pl) / sizeof (pl[0])) - 1)) {
                                /*
                                 * Page list won't fit in local array,
                                 * allocate one of the needed size.
                                 */
                                pl_alloc_sz =
                                    (btop(len) + 1) * sizeof (page_t *);
                                plp = kmem_alloc(pl_alloc_sz, KM_SLEEP);
                                plp[0] = NULL;
                                plsz = len;
                        } else if (rw == S_WRITE && svd->type == MAP_PRIVATE ||
                            svd->tr_state == SEGVN_TR_ON || rw == S_OTHER ||
                            (((size_t)(addr + PAGESIZE) <
                            (size_t)(seg->s_base + seg->s_size)) &&
                            hat_probe(as->a_hat, addr + PAGESIZE))) {
                                /*
                                 * Ask VOP_GETPAGE to return the exact number
                                 * of pages if
                                 * (a) this is a COW fault, or
                                 * (b) this is a software fault, or
                                 * (c) next page is already mapped.
                                 */
                                plsz = len;
                        } else {
                                /*
                                 * Ask VOP_GETPAGE to return adjacent pages
                                 * within the segment.
                                 */
                                plsz = MIN((size_t)FAULT_TMP_PAGES_SZ, (size_t)
                                plsz = MIN((size_t)PVN_GETPAGE_SZ, (size_t)
                                    ((seg->s_base + seg->s_size) - addr));
                                ASSERT((addr + plsz) <=
                                    (seg->s_base + seg->s_size));
                        }

                        /*
                         * Need to get some non-anonymous pages.
                         * We need to make only one call to GETPAGE to do
                         * this to prevent certain deadlocking conditions
                         * when we are doing locking.  In this case
                         * non_anon() should have picked up the smallest
                         * range which includes all the non-anonymous
                         * pages in the requested range.  We have to
                         * be careful regarding which rw flag to pass in
                         * because on a private mapping, the underlying
                         * object is never allowed to be written.
                         */
                        if (rw == S_WRITE && svd->type == MAP_PRIVATE) {
                                arw = S_READ;
                        } else {
                                arw = rw;
                        }
                        vp = svd->vp;
                        TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
                            "segvn_getpage:seg %p addr %p vp %p",
                            seg, addr, vp);
                        err = VOP_GETPAGE(vp, (offset_t)vp_off, vp_len,
                            &vpprot, plp, plsz, seg, addr + (vp_off - off), arw,
                            svd->cred, NULL);
                        if (err) {
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                segvn_pagelist_rele(plp);
                                if (pl_alloc_sz)
                                        kmem_free(plp, pl_alloc_sz);
                                return (FC_MAKE_ERR(err));
                        }
```

```
                        if (svd->type == MAP_PRIVATE)
                                vpprot &= ~PROT_WRITE;
                }
        }

        /*
         * N.B. at this time the plp array has all the needed non-anon
         * pages in addition to (possibly) having some adjacent pages.
         */

        /*
         * Always acquire the anon_array_lock to prevent
         * 2 threads from allocating separate anon slots for
         * the same "addr".
         *
         * If this is a copy-on-write fault and we don't already
         * have the anon_array_lock, acquire it to prevent the
         * fault routine from handling multiple copy-on-write faults
         * on the same "addr" in the same address space.
         *
         * Only one thread should deal with the fault since after
         * it is handled, the other threads can acquire a translation
         * to the newly created private page.  This prevents two or
         * more threads from creating different private pages for the
         * same fault.
         *
         * We grab "serialization" lock here if this is a MAP_PRIVATE segment
         * to prevent deadlock between this thread and another thread
         * which has soft-locked this page and wants to acquire serial_lock.
         * ( bug 4026339 )
         *
         * The fix for bug 4026339 becomes unnecessary when using the
         * locking scheme with per amp rwlock and a global set of hash
         * lock, anon_array_lock.  If we steal a vnode page when low
         * on memory and upgrad the page lock through page_rename,
         * then the page is PAGE_HANDLED, nothing needs to be done
         * for this page after returning from segvn_faultpage.
         *
         * But really, the page lock should be downgraded after
         * the stolen page is page_rename'd.
         */

        if (amp != NULL)
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

        /*
         * Ok, now loop over the address range and handle faults
         */
        for (a = addr; a < addr + len; a += PAGESIZE, off += PAGESIZE) {
                err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
                    type, rw, brkcow);
                if (err) {
                        if (amp != NULL)
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                        if (type == F_SOFTLOCK && a > addr) {
                                segvn_softunlock(seg, addr, (a - addr),
                                    S_OTHER);
                        }
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        segvn_pagelist_rele(plp);
                        if (pl_alloc_sz)
                                kmem_free(plp, pl_alloc_sz);
                        return (err);
                }
                if (vpage) {
                        vpage++;
                } else if (svd->vpage) {
                        page = seg_page(seg, addr);
                        vpage = &svd->vpage[++page];
                }
        }

        /* Didn't get pages from the underlying fs so we're done */
        if (!dogetpage)
                goto done;

        /*
         * Now handle any other pages in the list returned.
         * If the page can be used, load up the translations now.
         * Note that the for loop will only be entered if "plp"
         * is pointing to a non-NULL page pointer which means that
         * VOP_GETPAGE() was called and vpprot has been initialized.
         */
        if (svd->pageprot == 0)
                prot = svd->prot & vpprot;


        /*
         * Large Files: diff should be unsigned value because we started
         * supporting > 2GB segment sizes from 2.5.1 and when a
         * large file of size > 2GB gets mapped to address space
         * the diff value can be > 2GB.
         */

        for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
                size_t diff;
                struct anon *ap;
                int anon_index;
                anon_sync_obj_t cookie;
                int hat_flag = HAT_LOAD_ADV;

                if (svd->flags & MAP_TEXT) {
                        hat_flag |= HAT_LOAD_TEXT;
                }

                if (pp == PAGE_HANDLED)
                        continue;

                if (svd->tr_state != SEGVN_TR_ON &&
                    pp->p_offset >=  svd->offset &&
                    pp->p_offset < svd->offset + seg->s_size) {

                        diff = pp->p_offset - svd->offset;

                        /*
                         * Large Files: Following is the assertion
                         * validating the above cast.
                         */
                        ASSERT(svd->vp == pp->p_vnode);

                        page = btop(diff);
                        if (svd->pageprot)
                                prot = VPP_PROT(&svd->vpage[page]) & vpprot;

                        /*
                         * Prevent other threads in the address space from
                         * creating private pages (i.e., allocating anon slots)
                         * while we are in the process of loading translations
                         * to additional pages returned by the underlying
                         * object.
                         */
                        if (amp != NULL) {
```

```
                                anon_index = svd->anon_index + page;
                                anon_array_enter(amp, anon_index, &cookie);
                                ap = anon_get_ptr(amp->ahp, anon_index);
                        }
                        if ((amp == NULL) || (ap == NULL)) {
                                if (IS_VMODSORT(pp->p_vnode) ||
                                    enable_mbit_wa) {
                                        if (rw == S_WRITE)
                                                hat_setmod(pp);
                                        else if (rw != S_OTHER &&
                                            !hat_ismod(pp))
                                                prot &= ~PROT_WRITE;
                                }
                                /*
                                 * Skip mapping read ahead pages marked
                                 * for migration, so they will get migrated
                                 * properly on fault
                                 */
                                ASSERT(amp == NULL ||
                                    svd->rcookie == HAT_INVALID_REGION_COOKIE);
                                if ((prot & PROT_READ) && !PP_ISMIGRATE(pp)) {
                                        hat_memload_region(hat,
                                            seg->s_base + diff,
                                            pp, prot, hat_flag,
                                            svd->rcookie);
                                }
                        }
                        if (amp != NULL)
                                anon_array_exit(&cookie);
                }
                page_unlock(pp);
        }
done:
        if (amp != NULL)
                ANON_LOCK_EXIT(&amp->a_rwlock);
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        if (pl_alloc_sz)
                kmem_free(plp, pl_alloc_sz);
        return (0);
}
______unchanged_portion_omitted_
```