```
**********************************************************
    3232 Sat Jan 10 07:32:26 2015
new/usr/src/uts/common/vm/pvn.h
5508 move segvn #defines into seg_vn.c
Reviewed by: Marcel Telka <marcel@telka.sk>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#ifndef _VM_PVN_H
#define _VM_PVN_H

#include <sys/buf.h>
#include <vm/seg.h>

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef  _KERNEL

/*
 * VM - paged vnode.
 *
 * The VM system manages memory as a cache of paged vnodes.
 * This file desribes the interfaces to common subroutines
 * used to help implement the VM/file system routines.
 */

struct page     *pvn_read_kluster(struct vnode *vp, u_offset_t off,
                        struct seg *seg, caddr_t addr, u_offset_t *offp,
```

```
                        size_t *lenp, u_offset_t vp_off, size_t vp_len,
                        int isra);
struct page     *pvn_write_kluster(struct vnode *vp, struct page *pp,
                        u_offset_t *offp, size_t *lenp, u_offset_t vp_off,
                        size_t vp_len, int flags);
void            pvn_read_done(struct page *plist, int flags);
void            pvn_write_done(struct page *plist, int flags);
void            pvn_io_done(struct page *plist);
int             pvn_vplist_dirty(struct vnode *vp, u_offset_t off,
                        int (*putapage)(vnode_t *, struct page *, u_offset_t *,
                                size_t *, int, cred_t *),
                        int flags, struct cred *cred);
void            pvn_vplist_setdirty(vnode_t *vp, int (*page_check)(page_t *));
int             pvn_getdirty(struct page *pp, int flags);
void            pvn_vpzero(struct vnode *vp, u_offset_t vplen, size_t zbytes);
int             pvn_getpages(
                        int (*getpage)(vnode_t *, u_offset_t, size_t, uint_t *,
                                struct page *[], size_t, struct seg *,
                                caddr_t, enum seg_rw, cred_t *),
                        struct vnode *vp, u_offset_t off, size_t len,
                        uint_t *protp, struct page **pl, size_t plsz,
                        struct seg *seg, caddr_t addr, enum seg_rw rw,
                        struct cred *cred);
void            pvn_plist_init(struct page *pp, struct page **pl, size_t plsz,
                        u_offset_t off, size_t io_len, enum seg_rw rw);
void            pvn_init(void);

/*
 * The value is put in p_hash to identify marker pages. It is safe to
 * test p_hash ==(!=) PVN_VPLIST_HASH_TAG even without holding p_selock.
 */
#define PVN_VPLIST_HASH_TAG     ((page_t *)-1)

/*
 * When requesting pages from the getpage routines, pvn_getpages will
 * allocate space to return PVN_GETPAGE_NUM pages which map PVN_GETPAGE_SZ
 * worth of bytes.  These numbers are chosen to be the minimum of the max’s
 * given in terms of bytes and pages.
 */
#define PVN_MAX_GETPAGE_SZ      0x10000         /* getpage size limit */
#define PVN_MAX_GETPAGE_NUM     0x8             /* getpage page limit */

#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

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
  281355 Sat Jan 10 07:32:27 2015
new/usr/src/uts/common/vm/seg_vn.c
5508 move segvn #defines into seg_vn.c
Reviewed by: Marcel Telka <marcel@telka.sk>
**********************************************************
/*
 * 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 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

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

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - shared or copy-on-write from a vnode/anonymous memory.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/debug.h>
#include <sys/cred.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/bitmap.h>
#include <sys/swap.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/cmn_err.h>
#include <sys/callb.h>
```

```
#include <sys/vm.h>
#include <sys/dumphdr.h>
#include <sys/lgrp.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_vn.h>
#include <vm/pvn.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/vpage.h>
#include <sys/proc.h>
#include <sys/task.h>
#include <sys/project.h>
#include <sys/zone.h>
#include <sys/shm_impl.h>

/*
 * segvn_fault needs a temporary page list array.  To avoid calling kmem all
 * the time, it creates a small (PVN_GETPAGE_NUM entry) array and uses it if
 * it can.  In the rare case when this page list is not large enough, it
 * goes and gets a large enough array from kmem.
 *
 * This small page list array covers either 8 pages or 64kB worth of pages -
 * whichever is smaller.
 */
#define PVN_MAX_GETPAGE_SZ      0x10000
#define PVN_MAX_GETPAGE_NUM     0x8

#if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
#define PVN_GETPAGE_SZ  ptob(PVN_MAX_GETPAGE_NUM)
#define PVN_GETPAGE_NUM PVN_MAX_GETPAGE_NUM
#else
#define PVN_GETPAGE_SZ  PVN_MAX_GETPAGE_SZ
#define PVN_GETPAGE_NUM btop(PVN_MAX_GETPAGE_SZ)
#endif

#endif /* ! codereview */
/*
 * Private seg op routines.
 */
static int      segvn_dup(struct seg *seg, struct seg *newseg);
static int      segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
static void     segvn_free(struct seg *seg);
static faultcode_t segvn_fault(struct hat *hat, struct seg *seg,
                    caddr_t addr, size_t len, enum fault_type type,
                    enum seg_rw rw);
static faultcode_t segvn_faulta(struct seg *seg, caddr_t addr);
static int      segvn_setprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static int      segvn_checkprot(struct seg *seg, caddr_t addr,
                    size_t len, uint_t prot);
static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
static size_t   segvn_swapout(struct seg *seg);
static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
                    int attr, uint_t flags);
static size_t   segvn_incore(struct seg *seg, caddr_t addr, size_t len,
                    char *vec);
static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
                    int attr, int op, ulong_t *lockmap, size_t pos);
static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
                    uint_t *protv);
static u_offset_t       segvn_getoffset(struct seg *seg, caddr_t addr);
static int      segvn_gettype(struct seg *seg, caddr_t addr);
static int      segvn_getvp(struct seg *seg, caddr_t addr,
                    struct vnode **vpp);
static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,
                    uint_t behav);
static void     segvn_dump(struct seg *seg);
static int      segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
                    struct page ***ppp, enum lock_type type, enum seg_rw rw);
static int      segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
                    uint_t szc);
static int      segvn_getmemid(struct seg *seg, caddr_t addr,
                    memid_t *memidp);
static lgrp_mem_policy_info_t   *segvn_getpolicy(struct seg *, caddr_t);
static int      segvn_capable(struct seg *seg, segcapability_t capable);

struct  seg_ops segvn_ops = {
        segvn_dup,
        segvn_unmap,
        segvn_free,
        segvn_fault,
        segvn_faulta,
        segvn_setprot,
        segvn_checkprot,
        segvn_kluster,
        segvn_swapout,
        segvn_sync,
        segvn_incore,
        segvn_lockop,
        segvn_getprot,
        segvn_getoffset,
        segvn_gettype,
        segvn_getvp,
        segvn_advise,
        segvn_dump,
        segvn_pagelock,
        segvn_setpagesize,
        segvn_getmemid,
        segvn_getpolicy,
        segvn_capable,
};

/*
 * Common zfod structures, provided as a shorthand for others to use.
 */
static segvn_crargs_t zfod_segvn_crargs =
        SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
static segvn_crargs_t kzfod_segvn_crargs =
        SEGVN_ZFOD_ARGS(PROT_ZFOD & ~PROT_USER,
        PROT_ALL & ~PROT_USER);
static segvn_crargs_t stack_noexec_crargs =
        SEGVN_ZFOD_ARGS(PROT_ZFOD & ~PROT_EXEC, PROT_ALL);

caddr_t zfod_argsp = (caddr_t)&zfod_segvn_crargs;       /* user zfod argsp */
caddr_t kzfod_argsp = (caddr_t)&kzfod_segvn_crargs;     /* kernel zfod argsp */
caddr_t stack_exec_argsp = (caddr_t)&zfod_segvn_crargs; /* executable stack */
caddr_t stack_noexec_argsp = (caddr_t)&stack_noexec_crargs; /* noexec stack */

#define vpgtob(n)       ((n) * sizeof (struct vpage))   /* For brevity */

size_t  segvn_comb_thrshld = UINT_MAX;  /* patchable -- see 1196681 */

size_t  segvn_pglock_comb_thrshld = (1UL << 16);        /* 64K */
size_t  segvn_pglock_comb_balign = (1UL << 16);         /* 64K */
uint_t  segvn_pglock_comb_bshift;
size_t  segvn_pglock_comb_palign;

static int      segvn_concat(struct seg *, struct seg *, int);
static int      segvn_extend_prev(struct seg *, struct seg *,
```

```
                    struct segvn_crargs *, size_t);
static int      segvn_extend_next(struct seg *, struct seg *,
                    struct segvn_crargs *, size_t);
static void     segvn_softunlock(struct seg *, caddr_t, size_t, enum seg_rw);
static void     segvn_pagelist_rele(page_t **);
static void     segvn_setvnode_mpss(vnode_t *);
static void     segvn_relocate_pages(page_t **, page_t *);
static int      segvn_full_szcpages(page_t **, uint_t, int *, uint_t *);
static int      segvn_fill_vp_pages(struct segvn_data *, vnode_t *, u_offset_t,
    uint_t, page_t **, page_t **, uint_t *, int *);
static faultcode_t segvn_fault_vnodepages(struct hat *, struct seg *, caddr_t,
    caddr_t, enum fault_type, enum seg_rw, caddr_t, caddr_t, int);
static faultcode_t segvn_fault_anonpages(struct hat *, struct seg *, caddr_t,
    caddr_t, enum fault_type, enum seg_rw, caddr_t, caddr_t, int);
static faultcode_t segvn_faultpage(struct hat *, struct seg *, caddr_t,
    u_offset_t, struct vpage *, page_t **, uint_t,
    enum fault_type, enum seg_rw, int);
static void     segvn_vpage(struct seg *);
static size_t   segvn_count_swap_by_vpages(struct seg *);

static void segvn_purge(struct seg *seg);
static int segvn_reclaim(void *, caddr_t, size_t, struct page **,
    enum seg_rw, int);
static int shamp_reclaim(void *, caddr_t, size_t, struct page **,
    enum seg_rw, int);

static int sameprot(struct seg *, caddr_t, size_t);

static int segvn_demote_range(struct seg *, caddr_t, size_t, int, uint_t);
static int segvn_clrszc(struct seg *);
static struct seg *segvn_split_seg(struct seg *, caddr_t);
static int segvn_claim_pages(struct seg *, struct vpage *, u_offset_t,
    ulong_t, uint_t);

static void segvn_hat_rgn_unload_callback(caddr_t, caddr_t, caddr_t,
    size_t, void *, u_offset_t);

static struct kmem_cache *segvn_cache;
static struct kmem_cache **segvn_szc_cache;

#ifdef VM_STATS
static struct segvnvmstats_str {
        ulong_t fill_vp_pages[31];
        ulong_t fltvnpages[49];
        ulong_t fullszcpages[10];
        ulong_t relocatepages[3];
        ulong_t fltanpages[17];
        ulong_t pagelock[2];
        ulong_t demoterange[3];
} segvnvmstats;
#endif /* VM_STATS */

#define SDR_RANGE       1               /* demote entire range */
#define SDR_END         2               /* demote non aligned ends only */

#define CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr) {          \
                if ((len) != 0) {                                             \
                        lpgaddr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);  \
                        ASSERT(lpgaddr >= (seg)->s_base);                     \
                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)((addr) +    \
                            (len)), pgsz);                                    \
                        ASSERT(lpgeaddr > lpgaddr);                           \
                        ASSERT(lpgeaddr <= (seg)->s_base + (seg)->s_size);    \
                } else {                                                      \
                        lpgeaddr = lpgaddr = (addr);                          \
                }                                                             \
        }

/*ARGSUSED*/
static int
segvn_cache_constructor(void *buf, void *cdrarg, int kmflags)
{
        struct segvn_data *svd = buf;

        rw_init(&svd->lock, NULL, RW_DEFAULT, NULL);
        mutex_init(&svd->segfree_syncmtx, NULL, MUTEX_DEFAULT, NULL);
        svd->svn_trnext = svd->svn_trprev = NULL;
        return (0);
}

/*ARGSUSED1*/
static void
segvn_cache_destructor(void *buf, void *cdrarg)
{
        struct segvn_data *svd = buf;

        rw_destroy(&svd->lock);
        mutex_destroy(&svd->segfree_syncmtx);
}

/*ARGSUSED*/
static int
svntr_cache_constructor(void *buf, void *cdrarg, int kmflags)
{
        bzero(buf, sizeof (svntr_t));
        return (0);
}

/*
 * Patching this variable to non-zero allows the system to run with
 * stacks marked as "not executable".  It's a bit of a kludge, but is
 * provided as a tweakable for platforms that export those ABIs
 * (e.g. sparc V8) that have executable stacks enabled by default.
 * There are also some restrictions for platforms that don't actually
 * implement 'noexec' protections.
 *
 * Once enabled, the system is (therefore) unable to provide a fully
 * ABI-compliant execution environment, though practically speaking,
 * most everything works.  The exceptions are generally some interpreters
 * and debuggers that create executable code on the stack and jump
 * into it (without explicitly mprotecting the address range to include
 * PROT_EXEC).
 *
 * One important class of applications that are disabled are those
 * that have been transformed into malicious agents using one of the
 * numerous "buffer overflow" attacks.  See 4007890.
 */
int noexec_user_stack = 0;
int noexec_user_stack_log = 1;

int segvn_lpg_disable = 0;
uint_t segvn_maxpgszc = 0;

ulong_t segvn_vmpss_clrszc_cnt;
ulong_t segvn_vmpss_clrszc_err;
ulong_t segvn_fltvnpages_clrszc_cnt;
ulong_t segvn_fltvnpages_clrszc_err;
ulong_t segvn_setpgsz_align_err;
ulong_t segvn_setpgsz_anon_align_err;
ulong_t segvn_setpgsz_getattr_err;
ulong_t segvn_setpgsz_eof_err;
ulong_t segvn_faultvnmpss_align_err1;
```

```
ulong_t segvn_faultvnmpss_align_err2;
ulong_t segvn_faultvnmpss_align_err3;
ulong_t segvn_faultvnmpss_align_err4;
ulong_t segvn_faultvnmpss_align_err5;
ulong_t segvn_vmpss_pageio_deadlk_err;

int segvn_use_regions = 1;

/*
 * Segvn supports text replication optimization for NUMA platforms. Text
 * replica's are represented by anon maps (amp). There's one amp per text file
 * region per lgroup. A process chooses the amp for each of its text mappings
 * based on the lgroup assignment of its main thread (t_tid = 1). All
 * processes that want a replica on a particular lgroup for the same text file
 * mapping share the same amp. amp's are looked up in svntr_hashtab hash table
 * with vp,off,size,szc used as a key. Text replication segments are read only
 * MAP_PRIVATE|MAP_TEXT segments that map vnode. Replication is achieved by
 * forcing COW faults from vnode to amp and mapping amp pages instead of vnode
 * pages. Replication amp is assigned to a segment when it gets its first
 * pagefault. To handle main thread lgroup rehoming segvn_trasync_thread
 * rechecks periodically if the process still maps an amp local to the main
 * thread. If not async thread forces process to remap to an amp in the new
 * home lgroup of the main thread. Current text replication implementation
 * only provides the benefit to workloads that do most of their work in the
 * main thread of a process or all the threads of a process run in the same
 * lgroup. To extend text replication benefit to different types of
 * multithreaded workloads further work would be needed in the hat layer to
 * allow the same virtual address in the same hat to simultaneously map
 * different physical addresses (i.e. page table replication would be needed
 * for x86).
 *
 * amp pages are used instead of vnode pages as long as segment has a very
 * simple life cycle.  It's created via segvn_create(), handles S_EXEC
 * (S_READ) pagefaults and is fully unmapped.  If anything more complicated
 * happens such as protection is changed, real COW fault happens, pagesize is
 * changed, MC_LOCK is requested or segment is partially unmapped we turn off
 * text replication by converting the segment back to vnode only segment
 * (unmap segment's address range and set svd->amp to NULL).
 *
 * The original file can be changed after amp is inserted into
 * svntr_hashtab. Processes that are launched after the file is already
 * changed can't use the replica's created prior to the file change. To
 * implement this functionality hash entries are timestamped. Replica's can
 * only be used if current file modification time is the same as the timestamp
 * saved when hash entry was created. However just timestamps alone are not
 * sufficient to detect file modification via mmap(MAP_SHARED) mappings. We
 * deal with file changes via MAP_SHARED mappings differently. When writable
 * MAP_SHARED mappings are created to vnodes marked as executable we mark all
 * existing replica's for this vnode as not usable for future text
 * mappings. And we don't create new replica's for files that currently have
 * potentially writable MAP_SHARED mappings (i.e. vn_is_mapped(V_WRITE) is
 * true).
 */

#define SEGVN_TEXTREPL_MAXBYTES_FACTOR  (20)
size_t  segvn_textrepl_max_bytes_factor = SEGVN_TEXTREPL_MAXBYTES_FACTOR;

static ulong_t                  svntr_hashtab_sz = 512;
static svntr_bucket_t           *svntr_hashtab = NULL;
static struct kmem_cache        *svntr_cache;
static svntr_stats_t            *segvn_textrepl_stats;
static ksema_t                  segvn_trasync_sem;

int                             segvn_disable_textrepl = 1;
size_t                          textrepl_size_thresh = (size_t)-1;
size_t                          segvn_textrepl_bytes = 0;
size_t                          segvn_textrepl_max_bytes = 0;
clock_t                         segvn_update_textrepl_interval = 0;
int                             segvn_update_tr_time = 10;
int                             segvn_disable_textrepl_update = 0;

static void segvn_textrepl(struct seg *);
static void segvn_textunrepl(struct seg *, int);
static void segvn_inval_trcache(vnode_t *);
static void segvn_trasync_thread(void);
static void segvn_trupdate_wakeup(void *);
static void segvn_trupdate(void);
static void segvn_trupdate_seg(struct seg *, segvn_data_t *, svntr_t *,
    ulong_t);

/*
 * Initialize segvn data structures
 */
void
segvn_init(void)
{
        uint_t maxszc;
        uint_t szc;
        size_t pgsz;

        segvn_cache = kmem_cache_create("segvn_cache",
            sizeof (struct segvn_data), 0,
            segvn_cache_constructor, segvn_cache_destructor, NULL,
            NULL, NULL, 0);

        if (segvn_lpg_disable == 0) {
                szc = maxszc = page_num_pagesizes() - 1;
                if (szc == 0) {
                        segvn_lpg_disable = 1;
                }
                if (page_get_pagesize(0) != PAGESIZE) {
                        panic("segvn_init: bad szc 0");
                        /*NOTREACHED*/
                }
                while (szc != 0) {
                        pgsz = page_get_pagesize(szc);
                        if (pgsz <= PAGESIZE || !IS_P2ALIGNED(pgsz, pgsz)) {
                                panic("segvn_init: bad szc %d", szc);
                                /*NOTREACHED*/
                        }
                        szc--;
                }
                if (segvn_maxpgszc == 0 || segvn_maxpgszc > maxszc)
                        segvn_maxpgszc = maxszc;
        }

        if (segvn_maxpgszc) {
                segvn_szc_cache = (struct kmem_cache **)kmem_alloc(
                    (segvn_maxpgszc + 1) * sizeof (struct kmem_cache *),
                    KM_SLEEP);
        }

        for (szc = 1; szc <= segvn_maxpgszc; szc++) {
                char    str[32];

                (void) sprintf(str, "segvn_szc_cache%d", szc);
                segvn_szc_cache[szc] = kmem_cache_create(str,
                    page_get_pagecnt(szc) * sizeof (page_t *), 0,
                    NULL, NULL, NULL, NULL, NULL, KMC_NODEBUG);
        }
```

```
        if (segvn_use_regions && !hat_supported(HAT_SHARED_REGIONS, NULL))
                segvn_use_regions = 0;

        /*
         * For now shared regions and text replication segvn support
         * are mutually exclusive. This is acceptable because
         * currently significant benefit from text replication was
         * only observed on AMD64 NUMA platforms (due to relatively
         * small L2$ size) and currently we don't support shared
         * regions on x86.
         */
        if (segvn_use_regions && !segvn_disable_textrepl) {
                segvn_disable_textrepl = 1;
        }

#if defined(_LP64)
        if (lgrp_optimizations() && textrepl_size_thresh != (size_t)-1 &&
            !segvn_disable_textrepl) {
                ulong_t i;
                size_t hsz = svntr_hashtab_sz * sizeof (svntr_bucket_t);

                svntr_cache = kmem_cache_create("svntr_cache",
                    sizeof (svntr_t), 0, svntr_cache_constructor, NULL,
                    NULL, NULL, NULL, 0);
                svntr_hashtab = kmem_zalloc(hsz, KM_SLEEP);
                for (i = 0; i < svntr_hashtab_sz; i++) {
                        mutex_init(&svntr_hashtab[i].tr_lock,  NULL,
                            MUTEX_DEFAULT, NULL);
                }
                segvn_textrepl_max_bytes = ptob(physmem) /
                    segvn_textrepl_max_bytes_factor;
                segvn_textrepl_stats = kmem_zalloc(NCPU *
                    sizeof (svntr_stats_t), KM_SLEEP);
                sema_init(&segvn_trasync_sem, 0, NULL, SEMA_DEFAULT, NULL);
                (void) thread_create(NULL, 0, segvn_trasync_thread,
                    NULL, 0, &p0, TS_RUN, minclsyspri);
        }
#endif

        if (!ISP2(segvn_pglock_comb_balign) ||
            segvn_pglock_comb_balign < PAGESIZE) {
                segvn_pglock_comb_balign = 1UL << 16; /* 64K */
        }
        segvn_pglock_comb_bshift = highbit(segvn_pglock_comb_balign) - 1;
        segvn_pglock_comb_palign = btop(segvn_pglock_comb_balign);
}

#define SEGVN_PAGEIO    ((void *)0x1)
#define SEGVN_NOPAGEIO  ((void *)0x2)

static void
segvn_setvnode_mpss(vnode_t *vp)
{
        int err;

        ASSERT(vp->v_mpssdata == NULL ||
            vp->v_mpssdata == SEGVN_PAGEIO ||
            vp->v_mpssdata == SEGVN_NOPAGEIO);

        if (vp->v_mpssdata == NULL) {
                if (vn_vmpss_usepageio(vp)) {
                        err = VOP_PAGEIO(vp, (page_t *)NULL,
                            (u_offset_t)0, 0, 0, CRED(), NULL);
                } else {
                        err = ENOSYS;
                }
```

```c
                /*
                 * set v_mpssdata just once per vnode life
                 * so that it never changes.
                 */
                mutex_enter(&vp->v_lock);
                if (vp->v_mpssdata == NULL) {
                        if (err == EINVAL) {
                                vp->v_mpssdata = SEGVN_PAGEIO;
                        } else {
                                vp->v_mpssdata = SEGVN_NOPAGEIO;
                        }
                }
                mutex_exit(&vp->v_lock);
        }
}

int
segvn_create(struct seg *seg, void *argsp)
{
        struct segvn_crargs *a = (struct segvn_crargs *)argsp;
        struct segvn_data *svd;
        size_t swresv = 0;
        struct cred *cred;
        struct anon_map *amp;
        int error = 0;
        size_t pgsz;
        lgrp_mem_policy_t mpolicy = LGRP_MEM_POLICY_DEFAULT;
        int use_rgn = 0;
        int trok = 0;

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

        if (a->type != MAP_PRIVATE && a->type != MAP_SHARED) {
                panic("segvn_create type");
                /*NOTREACHED*/
        }

        /*
         * Check arguments.  If a shared anon structure is given then
         * it is illegal to also specify a vp.
         */
        if (a->amp != NULL && a->vp != NULL) {
                panic("segvn_create anon_map");
                /*NOTREACHED*/
        }

        if (a->type == MAP_PRIVATE && (a->flags & MAP_TEXT) &&
            a->vp != NULL && a->prot == (PROT_USER | PROT_READ | PROT_EXEC) &&
            segvn_use_regions) {
                use_rgn = 1;
        }

        /* MAP_NORESERVE on a MAP_SHARED segment is meaningless. */
        if (a->type == MAP_SHARED)
                a->flags &= ~MAP_NORESERVE;

        if (a->szc != 0) {
                if (segvn_lpg_disable != 0 || (a->szc == AS_MAP_NO_LPOOB) ||
                    (a->amp != NULL && a->type == MAP_PRIVATE) ||
                    (a->flags & MAP_NORESERVE) || seg->s_as == &kas) {
                        a->szc = 0;
                } else {
                        if (a->szc > segvn_maxpgszc)
                                a->szc = segvn_maxpgszc;
                        pgsz = page_get_pagesize(a->szc);
                        if (!IS_P2ALIGNED(seg->s_base, pgsz) ||
```

```c
                            !IS_P2ALIGNED(seg->s_size, pgsz)) {
                                a->szc = 0;
                        } else if (a->vp != NULL) {
                                if (IS_SWAPFSVP(a->vp) || VN_ISKAS(a->vp)) {
                                        /*
                                         * paranoid check.
                                         * hat_page_demote() is not supported
                                         * on swapfs pages.
                                         */
                                        a->szc = 0;
                                } else if (map_addr_vacalign_check(seg->s_base,
                                    a->offset & PAGEMASK)) {
                                        a->szc = 0;
                                }
                        } else if (a->amp != NULL) {
                                pgcnt_t anum = btopr(a->offset);
                                pgcnt_t pgcnt = page_get_pagecnt(a->szc);
                                if (!IS_P2ALIGNED(anum, pgcnt)) {
                                        a->szc = 0;
                                }
                        }
                }
        }

        /*
         * If segment may need private pages, reserve them now.
         */
        if (!(a->flags & MAP_NORESERVE) && ((a->vp == NULL && a->amp == NULL) ||
            (a->type == MAP_PRIVATE && (a->prot & PROT_WRITE)))) {
                if (anon_resv_zone(seg->s_size,
                    seg->s_as->a_proc->p_zone) == 0)
                        return (EAGAIN);
                swresv = seg->s_size;
                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
                    seg, swresv, 1);
        }

        /*
         * Reserve any mapping structures that may be required.
         *
         * Don’t do it for segments that may use regions. It’s currently a
         * noop in the hat implementations anyway.
         */
        if (!use_rgn) {
                hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);
        }

        if (a->cred) {
                cred = a->cred;
                crhold(cred);
        } else {
                crhold(cred = CRED());
        }

        /* Inform the vnode of the new mapping */
        if (a->vp != NULL) {
                error = VOP_ADDMAP(a->vp, a->offset & PAGEMASK,
                    seg->s_as, seg->s_base, seg->s_size, a->prot,
                    a->maxprot, a->type, cred, NULL);
                if (error) {
                        if (swresv != 0) {
                                anon_unresv_zone(swresv,
                                    seg->s_as->a_proc->p_zone);
                                TRACE_3(TR_FAC_VM, TR_ANON_PROC,
                                    "anon proc:%p %lu %u", seg, swresv, 0);
                        }
                        crfree(cred);
                        if (!use_rgn) {
                                hat_unload(seg->s_as->a_hat, seg->s_base,
                                    seg->s_size, HAT_UNLOAD_UNMAP);
                        }
                        return (error);
                }
                /*
                 * svntr_hashtab will be NULL if we support shared regions.
                 */
                trok = ((a->flags & MAP_TEXT) &&
                    (seg->s_size > textrepl_size_thresh ||
                    (a->flags & _MAP_TEXTREPL)) &&
                    lgrp_optimizations() && svntr_hashtab != NULL &&
                    a->type == MAP_PRIVATE && swresv == 0 &&
                    !(a->flags & MAP_NORESERVE) &&
                    seg->s_as != &kas && a->vp->v_type == VREG);

                ASSERT(!trok || !use_rgn);
        }

        /*
         * MAP_NORESERVE mappings don’t count towards the VSZ of a process
         * until we fault the pages in.
         */
        if ((a->vp == NULL || a->vp->v_type != VREG) &&
            a->flags & MAP_NORESERVE) {
                seg->s_as->a_resvsize -= seg->s_size;
        }

        /*
         * If more than one segment in the address space, and they’re adjacent
         * virtually, try to concatenate them.  Don’t concatenate if an
         * explicit anon_map structure was supplied (e.g., SystemV shared
         * memory) or if we’ll use text replication for this segment.
         */
        if (a->amp == NULL && !use_rgn && !trok) {
                struct seg *pseg, *nseg;
                struct segvn_data *psvd, *nsvd;
                lgrp_mem_policy_t ppolicy, npolicy;
                uint_t  lgrp_mem_policy_flags = 0;
                extern lgrp_mem_policy_t lgrp_mem_default_policy;

                /*
                 * Memory policy flags (lgrp_mem_policy_flags) is valid when
                 * extending stack/heap segments.
                 */
                if ((a->vp == NULL) && (a->type == MAP_PRIVATE) &&
                    !(a->flags & MAP_NORESERVE) && (seg->s_as != &kas)) {
                        lgrp_mem_policy_flags = a->lgrp_mem_policy_flags;
                } else {
                        /*
                         * Get policy when not extending it from another segment
                         */
                        mpolicy = lgrp_mem_policy_default(seg->s_size, a->type);
                }

                /*
                 * First, try to concatenate the previous and new segments
                 */
                pseg = AS_SEGPREV(seg->s_as, seg);
                if (pseg != NULL &&
                    pseg->s_base + pseg->s_size == seg->s_base &&
                    pseg->s_ops == &segvn_ops) {
                        /*
                         * Get memory allocation policy from previous segment.
```

```
                         * When extension is specified (e.g. for heap) apply
                         * this policy to the new segment regardless of the
                         * outcome of segment concatenation.  Extension occurs
                         * for non-default policy otherwise default policy is
                         * used and is based on extended segment size.
                         */
                        psvd = (struct segvn_data *)pseg->s_data;
                        ppolicy = psvd->policy_info.mem_policy;
                        if (lgrp_mem_policy_flags ==
                            LGRP_MP_FLAG_EXTEND_UP) {
                                if (ppolicy != lgrp_mem_default_policy) {
                                        mpolicy = ppolicy;
                                } else {
                                        mpolicy = lgrp_mem_policy_default(
                                            pseg->s_size + seg->s_size,
                                            a->type);
                                }
                        }

                        if (mpolicy == ppolicy &&
                            (pseg->s_size + seg->s_size <=
                            segvn_comb_thrshld || psvd->amp == NULL) &&
                            segvn_extend_prev(pseg, seg, a, swresv) == 0) {
                                /*
                                 * success! now try to concatenate
                                 * with following seg
                                 */
                                crfree(cred);
                                nseg = AS_SEGNEXT(pseg->s_as, pseg);
                                if (nseg != NULL &&
                                    nseg != pseg &&
                                    nseg->s_ops == &segvn_ops &&
                                    pseg->s_base + pseg->s_size ==
                                    nseg->s_base)
                                        (void) segvn_concat(pseg, nseg, 0);
                                ASSERT(pseg->s_szc == 0 ||
                                    (a->szc == pseg->s_szc &&
                                    IS_P2ALIGNED(pseg->s_base, pgsz) &&
                                    IS_P2ALIGNED(pseg->s_size, pgsz)));
                                return (0);
                        }
                }

                /*
                 * Failed, so try to concatenate with following seg
                 */
                nseg = AS_SEGNEXT(seg->s_as, seg);
                if (nseg != NULL &&
                    seg->s_base + seg->s_size == nseg->s_base &&
                    nseg->s_ops == &segvn_ops) {
                        /*
                         * Get memory allocation policy from next segment.
                         * When extension is specified (e.g. for stack) apply
                         * this policy to the new segment regardless of the
                         * outcome of segment concatenation.  Extension occurs
                         * for non-default policy otherwise default policy is
                         * used and is based on extended segment size.
                         */
                        nsvd = (struct segvn_data *)nseg->s_data;
                        npolicy = nsvd->policy_info.mem_policy;
                        if (lgrp_mem_policy_flags ==
                            LGRP_MP_FLAG_EXTEND_DOWN) {
                                if (npolicy != lgrp_mem_default_policy) {
                                        mpolicy = npolicy;
                                } else {
                                        mpolicy = lgrp_mem_policy_default(
                                            nseg->s_size + seg->s_size,
                                            a->type);
                                }
                        }

                        if (mpolicy == npolicy &&
                            segvn_extend_next(seg, nseg, a, swresv) == 0) {
                                crfree(cred);
                                ASSERT(nseg->s_szc == 0 ||
                                    (a->szc == nseg->s_szc &&
                                    IS_P2ALIGNED(nseg->s_base, pgsz) &&
                                    IS_P2ALIGNED(nseg->s_size, pgsz)));
                                return (0);
                        }
                }
        }

        if (a->vp != NULL) {
                VN_HOLD(a->vp);
                if (a->type == MAP_SHARED)
                        lgrp_shm_policy_init(NULL, a->vp);
        }
        svd = kmem_cache_alloc(segvn_cache, KM_SLEEP);

        seg->s_ops = &segvn_ops;
        seg->s_data = (void *)svd;
        seg->s_szc = a->szc;

        svd->seg = seg;
        svd->vp = a->vp;
        /*
         * Anonymous mappings have no backing file so the offset is meaningless.
         */
        svd->offset = a->vp ? (a->offset & PAGEMASK) : 0;
        svd->prot = a->prot;
        svd->maxprot = a->maxprot;
        svd->pageprot = 0;
        svd->type = a->type;
        svd->vpage = NULL;
        svd->cred = cred;
        svd->advice = MADV_NORMAL;
        svd->pageadvice = 0;
        svd->flags = (ushort_t)a->flags;
        svd->softlockcnt = 0;
        svd->softlockcnt_sbase = 0;
        svd->softlockcnt_send = 0;
        svd->rcookie = HAT_INVALID_REGION_COOKIE;
        svd->pageswap = 0;

        if (a->szc != 0 && a->vp != NULL) {
                segvn_setvnode_mpss(a->vp);
        }
        if (svd->type == MAP_SHARED && svd->vp != NULL &&
            (svd->vp->v_flag & VVMEXEC) && (svd->prot & PROT_WRITE)) {
                ASSERT(vn_is_mapped(svd->vp, V_WRITE));
                segvn_inval_trcache(svd->vp);
        }

        amp = a->amp;
        if ((svd->amp = amp) == NULL) {
                svd->anon_index = 0;
                if (svd->type == MAP_SHARED) {
                        svd->swresv = 0;
                        /*
                         * Shared mappings to a vp need no other setup.
                         * If we have a shared mapping to an anon_map object
```

```
                         * which hasn’t been allocated yet,  allocate the
                         * struct now so that it will be properly shared
                         * by remembering the swap reservation there.
                         */
                        if (a->vp == NULL) {
                                svd->amp = anonmap_alloc(seg->s_size, swresv,
                                    ANON_SLEEP);
                                svd->amp->a_szc = seg->s_szc;
                        }
                } else {
                        /*
                         * Private mapping (with or without a vp).
                         * Allocate anon_map when needed.
                         */
                        svd->swresv = swresv;
                }
        } else {
                pgcnt_t anon_num;

                /*
                 * Mapping to an existing anon_map structure without a vp.
                 * For now we will insure that the segment size isn’t larger
                 * than the size - offset gives us.  Later on we may wish to
                 * have the anon array dynamically allocated itself so that
                 * we don’t always have to allocate all the anon pointer slots.
                 * This of course involves adding extra code to check that we
                 * aren’t trying to use an anon pointer slot beyond the end
                 * of the currently allocated anon array.
                 */
                if ((amp->size - a->offset) < seg->s_size) {
                        panic("segvn_create anon_map size");
                        /*NOTREACHED*/
                }

                anon_num = btopr(a->offset);

                if (a->type == MAP_SHARED) {
                        /*
                         * SHARED mapping to a given anon_map.
                         */
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        amp->refcnt++;
                        if (a->szc > amp->a_szc) {
                                amp->a_szc = a->szc;
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        svd->anon_index = anon_num;
                        svd->swresv = 0;
                } else {
                        /*
                         * PRIVATE mapping to a given anon_map.
                         * Make sure that all the needed anon
                         * structures are created (so that we will
                         * share the underlying pages if nothing
                         * is written by this mapping) and then
                         * duplicate the anon array as is done
                         * when a privately mapped segment is dup’ed.
                         */
                        struct anon *ap;
                        caddr_t addr;
                        caddr_t eaddr;
                        ulong_t anon_idx;
                        int hat_flag = HAT_LOAD;

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

                        svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
                        svd->amp->a_szc = seg->s_szc;
                        svd->anon_index = 0;
                        svd->swresv = swresv;

                        /*
                         * Prevent 2 threads from allocating anon
                         * slots simultaneously.
                         */
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        eaddr = seg->s_base + seg->s_size;

                        for (anon_idx = anon_num, addr = seg->s_base;
                            addr < eaddr; addr += PAGESIZE, anon_idx++) {
                                page_t *pp;

                                if ((ap = anon_get_ptr(amp->ahp,
                                    anon_idx)) != NULL)
                                        continue;

                                /*
                                 * Allocate the anon struct now.
                                 * Might as well load up translation
                                 * to the page while we're at it...
                                 */
                                pp = anon_zero(seg, addr, &ap, cred);
                                if (ap == NULL || pp == NULL) {
                                        panic("segvn_create anon_zero");
                                        /*NOTREACHED*/
                                }

                                /*
                                 * Re-acquire the anon_map lock and
                                 * initialize the anon array entry.
                                 */
                                ASSERT(anon_get_ptr(amp->ahp,
                                    anon_idx) == NULL);
                                (void) anon_set_ptr(amp->ahp, anon_idx, ap,
                                    ANON_SLEEP);

                                ASSERT(seg->s_szc == 0);
                                ASSERT(!IS_VMODSORT(pp->p_vnode));

                                ASSERT(use_rgn == 0);
                                hat_memload(seg->s_as->a_hat, addr, pp,
                                    svd->prot & ~PROT_WRITE, hat_flag);

                                page_unlock(pp);
                        }
                        ASSERT(seg->s_szc == 0);
                        anon_dup(amp->ahp, anon_num, svd->amp->ahp,
                            0, seg->s_size);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }
        }

        /*
         * Set default memory allocation policy for segment
         *
         * Always set policy for private memory at least for initialization
         * even if this is a shared memory segment
         */
        (void) lgrp_privm_policy_set(mpolicy, &svd->policy_info, seg->s_size);
```

```
        if (svd->type == MAP_SHARED)
                (void) lgrp_shm_policy_set(mpolicy, svd->amp, svd->anon_index,
                    svd->vp, svd->offset, seg->s_size);

        if (use_rgn) {
                ASSERT(!trok);
                ASSERT(svd->amp == NULL);
                svd->rcookie = hat_join_region(seg->s_as->a_hat, seg->s_base,
                    seg->s_size, (void *)svd->vp, svd->offset, svd->prot,
                    (uchar_t)seg->s_szc, segvn_hat_rgn_unload_callback,
                    HAT_REGION_TEXT);
        }

        ASSERT(!trok || !(svd->prot & PROT_WRITE));
        svd->tr_state = trok ? SEGVN_TR_INIT : SEGVN_TR_OFF;

        return (0);
}

/*
 * Concatenate two existing segments, if possible.
 * Return 0 on success, -1 if two segments are not compatible
 * or -2 on memory allocation failure.
 * If amp_cat == 1 then try and concat segments with anon maps
 */
static int
segvn_concat(struct seg *seg1, struct seg *seg2, int amp_cat)
{
        struct segvn_data *svd1 = seg1->s_data;
        struct segvn_data *svd2 = seg2->s_data;
        struct anon_map *amp1 = svd1->amp;
        struct anon_map *amp2 = svd2->amp;
        struct vpage *vpage1 = svd1->vpage;
        struct vpage *vpage2 = svd2->vpage, *nvpage = NULL;
        size_t size, nvpsize;
        pgcnt_t npages1, npages2;

        ASSERT(seg1->s_as && seg2->s_as && seg1->s_as == seg2->s_as);
        ASSERT(AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));
        ASSERT(seg1->s_ops == seg2->s_ops);

        if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie) ||
            HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
                return (-1);
        }

        /* both segments exist, try to merge them */
#define incompat(x)     (svd1->x != svd2->x)
        if (incompat(vp) || incompat(maxprot) ||
            (!svd1->pageadvice && !svd2->pageadvice && incompat(advice)) ||
            (!svd1->pageprot && !svd2->pageprot && incompat(prot)) ||
            incompat(type) || incompat(cred) || incompat(flags) ||
            seg1->s_szc != seg2->s_szc || incompat(policy_info.mem_policy) ||
            (svd2->softlockcnt > 0) || svd1->softlockcnt_send > 0)
                return (-1);
#undef incompat

        /*
         * vp == NULL implies zfod, offset doesn't matter
         */
        if (svd1->vp != NULL &&
            svd1->offset + seg1->s_size != svd2->offset) {
                return (-1);
        }

        /*
```

```
         * Don’t concatenate if either segment uses text replication.
         */
        if (svd1->tr_state != SEGVN_TR_OFF || svd2->tr_state != SEGVN_TR_OFF) {
                return (-1);
        }

        /*
         * Fail early if we’re not supposed to concatenate
         * segments with non NULL amp.
         */
        if (amp_cat == 0 && (amp1 != NULL || amp2 != NULL)) {
                return (-1);
        }

        if (svd1->vp == NULL && svd1->type == MAP_SHARED) {
                if (amp1 != amp2) {
                        return (-1);
                }
                if (amp1 != NULL && svd1->anon_index + btop(seg1->s_size) !=
                    svd2->anon_index) {
                        return (-1);
                }
                ASSERT(amp1 == NULL || amp1->refcnt >= 2);
        }

        /*
         * If either seg has vpages, create a new merged vpage array.
         */
        if (vpage1 != NULL || vpage2 != NULL) {
                struct vpage *vp, *evp;

                npages1 = seg_pages(seg1);
                npages2 = seg_pages(seg2);
                nvpsize = vpgtob(npages1 + npages2);

                if ((nvpage = kmem_zalloc(nvpsize, KM_NOSLEEP)) == NULL) {
                        return (-2);
                }

                if (vpage1 != NULL) {
                        bcopy(vpage1, nvpage, vpgtob(npages1));
                } else {
                        evp = nvpage + npages1;
                        for (vp = nvpage; vp < evp; vp++) {
                                VPP_SETPROT(vp, svd1->prot);
                                VPP_SETADVICE(vp, svd1->advice);
                        }
                }

                if (vpage2 != NULL) {
                        bcopy(vpage2, nvpage + npages1, vpgtob(npages2));
                } else {
                        evp = nvpage + npages1 + npages2;
                        for (vp = nvpage + npages1; vp < evp; vp++) {
                                VPP_SETPROT(vp, svd2->prot);
                                VPP_SETADVICE(vp, svd2->advice);
                        }
                }

                if (svd2->pageswap && (!svd1->pageswap && svd1->swresv)) {
                        ASSERT(svd1->swresv == seg1->s_size);
                        ASSERT(!(svd1->flags & MAP_NORESERVE));
                        ASSERT(!(svd2->flags & MAP_NORESERVE));
                        evp = nvpage + npages1;
                        for (vp = nvpage; vp < evp; vp++) {
                                VPP_SETSWAPRES(vp);
```

```
                        }
                }

                if (svd1->pageswap && (!svd2->pageswap && svd2->swresv)) {
                        ASSERT(svd2->swresv == seg2->s_size);
                        ASSERT(!(svd1->flags & MAP_NORESERVE));
                        ASSERT(!(svd2->flags & MAP_NORESERVE));
                        vp = nvpage + npages1;
                        evp = vp + npages2;
                        for (; vp < evp; vp++) {
                                VPP_SETSWAPRES(vp);
                        }
                }
        }
        ASSERT((vpage1 != NULL || vpage2 != NULL) ||
            (svd1->pageswap == 0 && svd2->pageswap == 0));

        /*
         * If either segment has private pages, create a new merged anon
         * array. If mergeing shared anon segments just decrement anon map’s
         * refcnt.
         */
        if (amp1 != NULL && svd1->type == MAP_SHARED) {
                ASSERT(amp1 == amp2 && svd1->vp == NULL);
                ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
                ASSERT(amp1->refcnt >= 2);
                amp1->refcnt--;
                ANON_LOCK_EXIT(&amp1->a_rwlock);
                svd2->amp = NULL;
        } else if (amp1 != NULL || amp2 != NULL) {
                struct anon_hdr *nahp;
                struct anon_map *namp = NULL;
                size_t asize;

                ASSERT(svd1->type == MAP_PRIVATE);

                asize = seg1->s_size + seg2->s_size;
                if ((nahp = anon_create(btop(asize), ANON_NOSLEEP)) == NULL) {
                        if (nvpage != NULL) {
                                kmem_free(nvpage, nvpsize);
                        }
                        return (-2);
                }
                if (amp1 != NULL) {
                        /*
                         * XXX anon rwlock is not really needed because
                         * this is a private segment and we are writers.
                         */
                        ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
                        ASSERT(amp1->refcnt == 1);
                        if (anon_copy_ptr(amp1->ahp, svd1->anon_index,
                            nahp, 0, btop(seg1->s_size), ANON_NOSLEEP)) {
                                anon_release(nahp, btop(asize));
                                ANON_LOCK_EXIT(&amp1->a_rwlock);
                                if (nvpage != NULL) {
                                        kmem_free(nvpage, nvpsize);
                                }
                                return (-2);
                        }
                }
                if (amp2 != NULL) {
                        ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
                        ASSERT(amp2->refcnt == 1);
                        if (anon_copy_ptr(amp2->ahp, svd2->anon_index,
                            nahp, btop(seg1->s_size), btop(seg2->s_size),
                            ANON_NOSLEEP)) {
                                anon_release(nahp, btop(asize));
                                ANON_LOCK_EXIT(&amp2->a_rwlock);
                                if (amp1 != NULL) {
                                        ANON_LOCK_EXIT(&amp1->a_rwlock);
                                }
                                if (nvpage != NULL) {
                                        kmem_free(nvpage, nvpsize);
                                }
                                return (-2);
                        }
                }
                if (amp1 != NULL) {
                        namp = amp1;
                        anon_release(amp1->ahp, btop(amp1->size));
                }
                if (amp2 != NULL) {
                        if (namp == NULL) {
                                ASSERT(amp1 == NULL);
                                namp = amp2;
                                anon_release(amp2->ahp, btop(amp2->size));
                        } else {
                                amp2->refcnt--;
                                ANON_LOCK_EXIT(&amp2->a_rwlock);
                                anonmap_free(amp2);
                        }
                        svd2->amp = NULL; /* needed for seg_free */
                }
                namp->ahp = nahp;
                namp->size = asize;
                svd1->amp = namp;
                svd1->anon_index = 0;
                ANON_LOCK_EXIT(&namp->a_rwlock);
        }
        /*
         * Now free the old vpage structures.
         */
        if (nvpage != NULL) {
                if (vpage1 != NULL) {
                        kmem_free(vpage1, vpgtob(npages1));
                }
                if (vpage2 != NULL) {
                        svd2->vpage = NULL;
                        kmem_free(vpage2, vpgtob(npages2));
                }
                if (svd2->pageprot) {
                        svd1->pageprot = 1;
                }
                if (svd2->pageadvice) {
                        svd1->pageadvice = 1;
                }
                if (svd2->pageswap) {
                        svd1->pageswap = 1;
                }
                svd1->vpage = nvpage;
        }

        /* all looks ok, merge segments */
        svd1->swresv += svd2->swresv;
        svd2->swresv = 0;  /* so seg_free doesn’t release swap space */
        size = seg2->s_size;
        seg_free(seg2);
        seg1->s_size += size;
        return (0);
}

/*
```

```
 * Extend the previous segment (seg1) to include the
 * new segment (seg2 + a), if possible.
 * Return 0 on success.
 */
static int
segvn_extend_prev(seg1, seg2, a, swresv)
        struct seg *seg1, *seg2;
        struct segvn_crargs *a;
        size_t swresv;
{
        struct segvn_data *svd1 = (struct segvn_data *)seg1->s_data;
        size_t size;
        struct anon_map *amp1;
        struct vpage *new_vpage;

        /*
         * We don’t need any segment level locks for "segvn" data
         * since the address space is "write" locked.
         */
        ASSERT(seg1->s_as && AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));

        if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie)) {
                return (-1);
        }

        /* second segment is new, try to extend first */
        /* XXX - should also check cred */
        if (svd1->vp != a->vp || svd1->maxprot != a->maxprot ||
            (!svd1->pageprot && (svd1->prot != a->prot)) ||
            svd1->type != a->type || svd1->flags != a->flags ||
            seg1->s_szc != a->szc || svd1->softlockcnt_send > 0)
                return (-1);

        /* vp == NULL implies zfod, offset doesn’t matter */
        if (svd1->vp != NULL &&
            svd1->offset + seg1->s_size != (a->offset & PAGEMASK))
                return (-1);

        if (svd1->tr_state != SEGVN_TR_OFF) {
                return (-1);
        }

        amp1 = svd1->amp;
        if (amp1) {
                pgcnt_t newpgs;

                /*
                 * Segment has private pages, can data structures
                 * be expanded?
                 *
                 * Acquire the anon_map lock to prevent it from changing,
                 * if it is shared.  This ensures that the anon_map
                 * will not change while a thread which has a read/write
                 * lock on an address space references it.
                 * XXX - Don’t need the anon_map lock at all if "refcnt"
                 * is 1.
                 *
                 * Can’t grow a MAP_SHARED segment with an anonmap because
                 * there may be existing anon slots where we want to extend
                 * the segment and we wouldn’t know what to do with them
                 * (e.g., for tmpfs right thing is to just leave them there,
                 * for /dev/zero they should be cleared out).
                 */
                if (svd1->type == MAP_SHARED)
                        return (-1);
                ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
                if (amp1->refcnt > 1) {
                        ANON_LOCK_EXIT(&amp1->a_rwlock);
                        return (-1);
                }
                newpgs = anon_grow(amp1->ahp, &svd1->anon_index,
                    btop(seg1->s_size), btop(seg2->s_size), ANON_NOSLEEP);

                if (newpgs == 0) {
                        ANON_LOCK_EXIT(&amp1->a_rwlock);
                        return (-1);
                }
                amp1->size = ptob(newpgs);
                ANON_LOCK_EXIT(&amp1->a_rwlock);
        }
        if (svd1->vpage != NULL) {
                struct vpage *vp, *evp;
                new_vpage =
                    kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
                        KM_NOSLEEP);
                if (new_vpage == NULL)
                        return (-1);
                bcopy(svd1->vpage, new_vpage, vpgtob(seg_pages(seg1)));
                kmem_free(svd1->vpage, vpgtob(seg_pages(seg1)));
                svd1->vpage = new_vpage;

                vp = new_vpage + seg_pages(seg1);
                evp = vp + seg_pages(seg2);
                for (; vp < evp; vp++)
                        VPP_SETPROT(vp, a->prot);
                if (svd1->pageswap && swresv) {
                        ASSERT(!(svd1->flags & MAP_NORESERVE));
                        ASSERT(swresv == seg2->s_size);
                        vp = new_vpage + seg_pages(seg1);
                        for (; vp < evp; vp++) {
                                VPP_SETSWAPRES(vp);
                        }
                }
        }
        ASSERT(svd1->vpage != NULL || svd1->pageswap == 0);
        size = seg2->s_size;
        seg_free(seg2);
        seg1->s_size += size;
        svd1->swresv += swresv;
        if (svd1->pageprot && (a->prot & PROT_WRITE) &&
            svd1->type == MAP_SHARED && svd1->vp != NULL &&
            (svd1->vp->v_flag & VVMEXEC)) {
                ASSERT(vn_is_mapped(svd1->vp, V_WRITE));
                segvn_inval_trcache(svd1->vp);
        }
        return (0);
}

/*
 * Extend the next segment (seg2) to include the
 * new segment (seg1 + a), if possible.
 * Return 0 on success.
 */
static int
segvn_extend_next(
        struct seg *seg1,
        struct seg *seg2,
        struct segvn_crargs *a,
        size_t swresv)
{
        struct segvn_data *svd2 = (struct segvn_data *)seg2->s_data;
```

```
        size_t size;
        struct anon_map *amp2;
        struct vpage *new_vpage;

        /*
         * We don't need any segment level locks for "segvn" data
         * since the address space is "write" locked.
         */
        ASSERT(seg2->s_as && AS_WRITE_HELD(seg2->s_as, &seg2->s_as->a_lock));

        if (HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
                return (-1);
        }

        /* first segment is new, try to extend second */
        /* XXX - should also check cred */
        if (svd2->vp != a->vp || svd2->maxprot != a->maxprot ||
            (!svd2->pageprot && (svd2->prot != a->prot)) ||
            svd2->type != a->type || svd2->flags != a->flags ||
            seg2->s_szc != a->szc || svd2->softlockcnt_sbase > 0)
                return (-1);
        /* vp == NULL implies zfod, offset doesn't matter */
        if (svd2->vp != NULL &&
            (a->offset & PAGEMASK) + seg1->s_size != svd2->offset)
                return (-1);

        if (svd2->tr_state != SEGVN_TR_OFF) {
                return (-1);
        }

        amp2 = svd2->amp;
        if (amp2) {
                pgcnt_t newpgs;

                /*
                 * Segment has private pages, can data structures
                 * be expanded?
                 *
                 * Acquire the anon_map lock to prevent it from changing,
                 * if it is shared.  This ensures that the anon_map
                 * will not change while a thread which has a read/write
                 * lock on an address space references it.
                 *
                 * XXX - Don't need the anon_map lock at all if "refcnt"
                 * is 1.
                 */
                if (svd2->type == MAP_SHARED)
                        return (-1);

                ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
                if (amp2->refcnt > 1) {
                        ANON_LOCK_EXIT(&amp2->a_rwlock);
                        return (-1);
                }
                newpgs = anon_grow(amp2->ahp, &svd2->anon_index,
                    btop(seg2->s_size), btop(seg1->s_size),
                    ANON_NOSLEEP | ANON_GROWDOWN);

                if (newpgs == 0) {
                        ANON_LOCK_EXIT(&amp2->a_rwlock);
                        return (-1);
                }
                amp2->size = ptob(newpgs);
                ANON_LOCK_EXIT(&amp2->a_rwlock);
        }
        if (svd2->vpage != NULL) {
                struct vpage *vp, *evp;
                new_vpage =
                    kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
                    KM_NOSLEEP);
                if (new_vpage == NULL) {
                        /* Not merging segments so adjust anon_index back */
                        if (amp2)
                                svd2->anon_index += seg_pages(seg1);
                        return (-1);
                }
                bcopy(svd2->vpage, new_vpage + seg_pages(seg1),
                    vpgtob(seg_pages(seg2)));
                kmem_free(svd2->vpage, vpgtob(seg_pages(seg2)));
                svd2->vpage = new_vpage;

                vp = new_vpage;
                evp = vp + seg_pages(seg1);
                for (; vp < evp; vp++)
                        VPP_SETPROT(vp, a->prot);
                if (svd2->pageswap && swresv) {
                        ASSERT(!(svd2->flags & MAP_NORESERVE));
                        ASSERT(swresv == seg1->s_size);
                        vp = new_vpage;
                        for (; vp < evp; vp++) {
                                VPP_SETSWAPRES(vp);
                        }
                }
        }
        ASSERT(svd2->vpage != NULL || svd2->pageswap == 0);
        size = seg1->s_size;
        seg_free(seg1);
        seg2->s_size += size;
        seg2->s_base -= size;
        svd2->offset -= size;
        svd2->swresv += swresv;
        if (svd2->pageprot && (a->prot & PROT_WRITE) &&
            svd2->type == MAP_SHARED && svd2->vp != NULL &&
            (svd2->vp->v_flag & VVMEXEC)) {
                ASSERT(vn_is_mapped(svd2->vp, V_WRITE));
                segvn_inval_trcache(svd2->vp);
        }
        return (0);
}

static int
segvn_dup(struct seg *seg, struct seg *newseg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct segvn_data *newsvd;
        pgcnt_t npages = seg_pages(seg);
        int error = 0;
        uint_t prot;
        size_t len;
        struct anon_map *amp;

        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(newseg->s_as->a_proc->p_parent == curproc);

        /*
         * If segment has anon reserved, reserve more for the new seg.
         * For a MAP_NORESERVE segment swresv will be a count of all the
         * allocated anon slots; thus we reserve for the child as many slots
         * as the parent has allocated. This semantic prevents the child or
         * parent from dieing during a copy-on-write fault caused by trying
         * to write a shared pre-existing anon page.
         */
```

```
        if ((len = svd->swresv) != 0) {
                if (anon_resv(svd->swresv) == 0)
                        return (ENOMEM);

                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
                    seg, len, 0);
        }

        newsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);

        newseg->s_ops = &segvn_ops;
        newseg->s_data = (void *)newsvd;
        newseg->s_szc = seg->s_szc;

        newsvd->seg = newseg;
        if ((newsvd->vp = svd->vp) != NULL) {
                VN_HOLD(svd->vp);
                if (svd->type == MAP_SHARED)
                        lgrp_shm_policy_init(NULL, svd->vp);
        }
        newsvd->offset = svd->offset;
        newsvd->prot = svd->prot;
        newsvd->maxprot = svd->maxprot;
        newsvd->pageprot = svd->pageprot;
        newsvd->type = svd->type;
        newsvd->cred = svd->cred;
        crhold(newsvd->cred);
        newsvd->advice = svd->advice;
        newsvd->pageadvice = svd->pageadvice;
        newsvd->swresv = svd->swresv;
        newsvd->pageswap = svd->pageswap;
        newsvd->flags = svd->flags;
        newsvd->softlockcnt = 0;
        newsvd->softlockcnt_sbase = 0;
        newsvd->softlockcnt_send = 0;
        newsvd->policy_info = svd->policy_info;
        newsvd->rcookie = HAT_INVALID_REGION_COOKIE;

        if ((amp = svd->amp) == NULL || svd->tr_state == SEGVN_TR_ON) {
                /*
                 * Not attaching to a shared anon object.
                 */
                ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie) ||
                    svd->tr_state == SEGVN_TR_OFF);
                if (svd->tr_state == SEGVN_TR_ON) {
                        ASSERT(newsvd->vp != NULL && amp != NULL);
                        newsvd->tr_state = SEGVN_TR_INIT;
                } else {
                        newsvd->tr_state = svd->tr_state;
                }
                newsvd->amp = NULL;
                newsvd->anon_index = 0;
        } else {
                /* regions for now are only used on pure vnode segments */
                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                newsvd->tr_state = SEGVN_TR_OFF;
                if (svd->type == MAP_SHARED) {
                        newsvd->amp = amp;
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        amp->refcnt++;
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        newsvd->anon_index = svd->anon_index;
                } else {
                        int reclaim = 1;
                        /*
                         * Allocate and initialize new anon_map structure.
                         */
                        newsvd->amp = anonmap_alloc(newseg->s_size, 0,
                            ANON_SLEEP);
                        newsvd->amp->a_szc = newseg->s_szc;
                        newsvd->anon_index = 0;

                        /*
                         * We don't have to acquire the anon_map lock
                         * for the new segment (since it belongs to an
                         * address space that is still not associated
                         * with any process), or the segment in the old
                         * address space (since all threads in it
                         * are stopped while duplicating the address space).
                         */

                        /*
                         * The goal of the following code is to make sure that
                         * softlocked pages do not end up as copy on write
                         * pages.  This would cause problems where one
                         * thread writes to a page that is COW and a different
                         * thread in the same process has softlocked it.  The
                         * softlock lock would move away from this process
                         * because the write would cause this process to get
                         * a copy (without the softlock).
                         *
                         * The strategy here is to just break the
                         * sharing on pages that could possibly be
                         * softlocked.
                         */
retry:
                        if (svd->softlockcnt) {
                                struct anon *ap, *newap;
                                size_t i;
                                uint_t vpprot;
                                page_t *anon_pl[1+1], *pp;
                                caddr_t addr;
                                ulong_t old_idx = svd->anon_index;
                                ulong_t new_idx = 0;

                                /*
                                 * The softlock count might be non zero
                                 * because some pages are still stuck in the
                                 * cache for lazy reclaim. Flush the cache
                                 * now. This should drop the count to zero.
                                 * [or there is really I/O going on to these
                                 * pages]. Note, we have the writers lock so
                                 * nothing gets inserted during the flush.
                                 */
                                if (reclaim == 1) {
                                        segvn_purge(seg);
                                        reclaim = 0;
                                        goto retry;
                                }
                                i = btopr(seg->s_size);
                                addr = seg->s_base;
                                /*
                                 * XXX break cow sharing using PAGESIZE
                                 * pages. They will be relocated into larger
                                 * pages at fault time.
                                 */
                                while (i-- > 0) {
                                        if (ap = anon_get_ptr(amp->ahp,
                                            old_idx)) {
                                                error = anon_getpage(&ap,
```

```
                                                    &vpprot, anon_pl, PAGESIZE,
                                                    seg, addr, S_READ,
                                                    svd->cred);
                                                if (error) {
                                                        newsvd->vpage = NULL;
                                                        goto out;
                                                }
                                                /*
                                                 * prot need not be computed
                                                 * below 'cause anon_private is
                                                 * going to ignore it anyway
                                                 * as child doesn't inherit
                                                 * pagelock from parent.
                                                 */
                                                prot = svd->pageprot ?
                                                    VPP_PROT(
                                                    &svd->vpage[
                                                    seg_page(seg, addr)])
                                                    : svd->prot;
                                                pp = anon_private(&newap,
                                                    newseg, addr, prot,
                                                    anon_pl[0], 0,
                                                    newsvd->cred);
                                                if (pp == NULL) {
                                                        /* no mem abort */
                                                        newsvd->vpage = NULL;
                                                        error = ENOMEM;
                                                        goto out;
                                                }
                                                (void) anon_set_ptr(
                                                    newsvd->amp->ahp, new_idx,
                                                    newap, ANON_SLEEP);
                                                page_unlock(pp);
                                        }
                                        addr += PAGESIZE;
                                        old_idx++;
                                        new_idx++;
                                }
                        } else {        /* common case */
                                if (seg->s_szc != 0) {
                                        /*
                                         * If at least one of anon slots of a
                                         * large page exists then make sure
                                         * all anon slots of a large page
                                         * exist to avoid partial cow sharing
                                         * of a large page in the future.
                                         */
                                        anon_dup_fill_holes(amp->ahp,
                                            svd->anon_index, newsvd->amp->ahp,
                                            0, seg->s_size, seg->s_szc,
                                            svd->vp != NULL);
                                } else {
                                        anon_dup(amp->ahp, svd->anon_index,
                                            newsvd->amp->ahp, 0, seg->s_size);
                                }

                                hat_clrattr(seg->s_as->a_hat, seg->s_base,
                                    seg->s_size, PROT_WRITE);
                        }
                }
        }
        /*
         * If necessary, create a vpage structure for the new segment.
         * Do not copy any page lock indications.
         */
        if (svd->vpage != NULL) {
                uint_t i;
                struct vpage *ovp = svd->vpage;
                struct vpage *nvp;

                nvp = newsvd->vpage =
                    kmem_alloc(vpgtob(npages), KM_SLEEP);
                for (i = 0; i < npages; i++) {
                        *nvp = *ovp++;
                        VPP_CLRPPLOCK(nvp++);
                }
        } else
                newsvd->vpage = NULL;

        /* Inform the vnode of the new mapping */
        if (newsvd->vp != NULL) {
                error = VOP_ADDMAP(newsvd->vp, (offset_t)newsvd->offset,
                    newseg->s_as, newseg->s_base, newseg->s_size, newsvd->prot,
                    newsvd->maxprot, newsvd->type, newsvd->cred, NULL);
        }
out:
        if (error == 0 && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(newsvd->amp == NULL);
                ASSERT(newsvd->tr_state == SEGVN_TR_OFF);
                newsvd->rcookie = svd->rcookie;
                hat_dup_region(newseg->s_as->a_hat, newsvd->rcookie);
        }
        return (error);
}


/*
 * callback function to invoke free_vp_pages() for only those pages actually
 * processed by the HAT when a shared region is destroyed.
 */
extern int free_pages;

static void
segvn_hat_rgn_unload_callback(caddr_t saddr, caddr_t eaddr, caddr_t r_saddr,
    size_t r_size, void *r_obj, u_offset_t r_objoff)
{
        u_offset_t off;
        size_t len;
        vnode_t *vp = (vnode_t *)r_obj;

        ASSERT(eaddr > saddr);
        ASSERT(saddr >= r_saddr);
        ASSERT(saddr < r_saddr + r_size);
        ASSERT(eaddr > r_saddr);
        ASSERT(eaddr <= r_saddr + r_size);
        ASSERT(vp != NULL);

        if (!free_pages) {
                return;
        }

        len = eaddr - saddr;
        off = (saddr - r_saddr) + r_objoff;
        free_vp_pages(vp, off, len);
}

/*
 * callback function used by segvn_unmap to invoke free_vp_pages() for only
 * those pages actually processed by the HAT
 */
static void
segvn_hat_unload_callback(hat_callback_t *cb)
```

```
{
        struct seg              *seg = cb->hcb_data;
        struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
        size_t                  len;
        u_offset_t              off;

        ASSERT(svd->vp != NULL);
        ASSERT(cb->hcb_end_addr > cb->hcb_start_addr);
        ASSERT(cb->hcb_start_addr >= seg->s_base);

        len = cb->hcb_end_addr - cb->hcb_start_addr;
        off = cb->hcb_start_addr - seg->s_base;
        free_vp_pages(svd->vp, svd->offset + off, len);
}

/*
 * This function determines the number of bytes of swap reserved by
 * a segment for which per-page accounting is present. It is used to
 * calculate the correct value of a segvn_data’s swresv.
 */
static size_t
segvn_count_swap_by_vpages(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vp, *evp;
        size_t nswappages = 0;

        ASSERT(svd->pageswap);
        ASSERT(svd->vpage != NULL);

        evp = &svd->vpage[seg_page(seg, seg->s_base + seg->s_size)];

        for (vp = svd->vpage; vp < evp; vp++) {
                if (VPP_ISSWAPRES(vp))
                        nswappages++;
        }

        return (nswappages << PAGESHIFT);
}

static int
segvn_unmap(struct seg *seg, caddr_t addr, size_t len)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct segvn_data *nsvd;
        struct seg *nseg;
        struct anon_map *amp;
        pgcnt_t opages;         /* old segment size in pages */
        pgcnt_t npages;         /* new segment size in pages */
        pgcnt_t dpages;         /* pages being deleted (unmapped) */
        hat_callback_t callback;        /* used for free_vp_pages() */
        hat_callback_t *cbp = NULL;
        caddr_t nbase;
        size_t nsize;
        size_t oswresv;
        int reclaim = 1;

        /*
         * We don’t need any segment level locks for "segvn" data
         * since the address space is "write" locked.
         */
        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

        /*
         * Fail the unmap if pages are SOFTLOCKed through this mapping.
         * softlockcnt is protected from change by the as write lock.
         */
retry:
        if (svd->softlockcnt > 0) {
                ASSERT(svd->tr_state == SEGVN_TR_OFF);

                /*
                 * If this is shared segment non 0 softlockcnt
                 * means locked pages are still in use.
                 */
                if (svd->type == MAP_SHARED) {
                        return (EAGAIN);
                }

                /*
                 * since we do have the writers lock nobody can fill
                 * the cache during the purge. The flush either succeeds
                 * or we still have pending I/Os.
                 */
                if (reclaim == 1) {
                        segvn_purge(seg);
                        reclaim = 0;
                        goto retry;
                }
                return (EAGAIN);
        }

        /*
         * Check for bad sizes
         */
        if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
            (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET)) {
                panic("segvn_unmap");
                /*NOTREACHED*/
        }

        if (seg->s_szc != 0) {
                size_t pgsz = page_get_pagesize(seg->s_szc);
                int err;
                if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
                        ASSERT(seg->s_base != addr || seg->s_size != len);
                        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                                ASSERT(svd->amp == NULL);
                                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                                hat_leave_region(seg->s_as->a_hat,
                                    svd->rcookie, HAT_REGION_TEXT);
                                svd->rcookie = HAT_INVALID_REGION_COOKIE;
                                /*
                                 * could pass a flag to segvn_demote_range()
                                 * below to tell it not to do any unloads but
                                 * this case is rare enough to not bother for
                                 * now.
                                 */
                        } else if (svd->tr_state == SEGVN_TR_INIT) {
                                svd->tr_state = SEGVN_TR_OFF;
                        } else if (svd->tr_state == SEGVN_TR_ON) {
                                ASSERT(svd->amp != NULL);
                                segvn_textunrepl(seg, 1);
                                ASSERT(svd->amp == NULL);
                                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                        }
                        VM_STAT_ADD(segvnvmstats.demoterange[0]);
                        err = segvn_demote_range(seg, addr, len, SDR_END, 0);
                        if (err == 0) {
                                return (IE_RETRY);
                        }
                        return (err);
```

```
                }
        }

        /* Inform the vnode of the unmapping. */
        if (svd->vp) {
                int error;

                error = VOP_DELMAP(svd->vp,
                    (offset_t)svd->offset + (uintptr_t)(addr - seg->s_base),
                    seg->s_as, addr, len, svd->prot, svd->maxprot,
                    svd->type, svd->cred, NULL);

                if (error == EAGAIN)
                        return (error);
        }

        /*
         * Remove any page locks set through this mapping.
         * If text replication is not off no page locks could have been
         * established via this mapping.
         */
        if (svd->tr_state == SEGVN_TR_OFF) {
                (void) segvn_lockop(seg, addr, len, 0, MC_UNLOCK, NULL, 0);
        }

        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(svd->amp == NULL);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                ASSERT(svd->type == MAP_PRIVATE);
                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                    HAT_REGION_TEXT);
                svd->rcookie = HAT_INVALID_REGION_COOKIE;
        } else if (svd->tr_state == SEGVN_TR_ON) {
                ASSERT(svd->amp != NULL);
                ASSERT(svd->pageprot == 0 && !(svd->prot & PROT_WRITE));
                segvn_textunrepl(seg, 1);
                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
        } else {
                if (svd->tr_state != SEGVN_TR_OFF) {
                        ASSERT(svd->tr_state == SEGVN_TR_INIT);
                        svd->tr_state = SEGVN_TR_OFF;
                }
                /*
                 * Unload any hardware translations in the range to be taken
                 * out. Use a callback to invoke free_vp_pages() effectively.
                 */
                if (svd->vp != NULL && free_pages != 0) {
                        callback.hcb_data = seg;
                        callback.hcb_function = segvn_hat_unload_callback;
                        cbp = &callback;
                }
                hat_unload_callback(seg->s_as->a_hat, addr, len,
                    HAT_UNLOAD_UNMAP, cbp);

                if (svd->type == MAP_SHARED && svd->vp != NULL &&
                    (svd->vp->v_flag & VVMEXEC) &&
                    ((svd->prot & PROT_WRITE) || svd->pageprot)) {
                        segvn_inval_trcache(svd->vp);
                }
        }

        /*
         * Check for entire segment
         */
        if (addr == seg->s_base && len == seg->s_size) {
                seg_free(seg);
                return (0);
        }

        opages = seg_pages(seg);
        dpages = btop(len);
        npages = opages - dpages;
        amp = svd->amp;
        ASSERT(amp == NULL || amp->a_szc >= seg->s_szc);

        /*
         * Check for beginning of segment
         */
        if (addr == seg->s_base) {
                if (svd->vpage != NULL) {
                        size_t nbytes;
                        struct vpage *ovpage;

                        ovpage = svd->vpage;    /* keep pointer to vpage */

                        nbytes = vpgtob(npages);
                        svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
                        bcopy(&ovpage[dpages], svd->vpage, nbytes);

                        /* free up old vpage */
                        kmem_free(ovpage, vpgtob(opages));
                }
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
                                /*
                                 * Shared anon map is no longer in use. Before
                                 * freeing its pages purge all entries from
                                 * pcache that belong to this amp.
                                 */
                                if (svd->type == MAP_SHARED) {
                                        ASSERT(amp->refcnt == 1);
                                        ASSERT(svd->softlockcnt == 0);
                                        anonmap_purge(amp);
                                }
                                /*
                                 * Free up now unused parts of anon_map array.
                                 */
                                if (amp->a_szc == seg->s_szc) {
                                        if (seg->s_szc != 0) {
                                                anon_free_pages(amp->ahp,
                                                    svd->anon_index, len,
                                                    seg->s_szc);
                                        } else {
                                                anon_free(amp->ahp,
                                                    svd->anon_index,
                                                    len);
                                        }
                                } else {
                                        ASSERT(svd->type == MAP_SHARED);
                                        ASSERT(amp->a_szc > seg->s_szc);
                                        anon_shmap_free_pages(amp,
                                            svd->anon_index, len);
                                }

                                /*
                                 * Unreserve swap space for the
                                 * unmapped chunk of this segment in
                                 * case it’s MAP_SHARED
                                 */
                                if (svd->type == MAP_SHARED) {
                                        anon_unresv_zone(len,
```

```
                                            seg->s_as->a_proc->p_zone);
                                        amp->swresv -= len;
                                }
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        svd->anon_index += dpages;
                }
                if (svd->vp != NULL)
                        svd->offset += len;

                seg->s_base += len;
                seg->s_size -= len;

                if (svd->swresv) {
                        if (svd->flags & MAP_NORESERVE) {
                                ASSERT(amp);
                                oswresv = svd->swresv;

                                svd->swresv = ptob(anon_pages(amp->ahp,
                                    svd->anon_index, npages));
                                anon_unresv_zone(oswresv - svd->swresv,
                                    seg->s_as->a_proc->p_zone);
                                if (SEG_IS_PARTIAL_RESV(seg))
                                        seg->s_as->a_resvsize -= oswresv -
                                            svd->swresv;
                        } else {
                                size_t unlen;

                                if (svd->pageswap) {
                                        oswresv = svd->swresv;
                                        svd->swresv =
                                            segvn_count_swap_by_vpages(seg);
                                        ASSERT(oswresv >= svd->swresv);
                                        unlen = oswresv - svd->swresv;
                                } else {
                                        svd->swresv -= len;
                                        ASSERT(svd->swresv == seg->s_size);
                                        unlen = len;
                                }
                                anon_unresv_zone(unlen,
                                    seg->s_as->a_proc->p_zone);
                        }
                        TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
                            seg, len, 0);
                }

                return (0);
        }

        /*
         * Check for end of segment
         */
        if (addr + len == seg->s_base + seg->s_size) {
                if (svd->vpage != NULL) {
                        size_t nbytes;
                        struct vpage *ovpage;

                        ovpage = svd->vpage;    /* keep pointer to vpage */

                        nbytes = vpgtob(npages);
                        svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
                        bcopy(ovpage, svd->vpage, nbytes);

                        /* free up old vpage */
                        kmem_free(ovpage, vpgtob(opages));
                }
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
                                /*
                                 * Free up now unused parts of anon_map array.
                                 */
                                ulong_t an_idx = svd->anon_index + npages;

                                /*
                                 * Shared anon map is no longer in use. Before
                                 * freeing its pages purge all entries from
                                 * pcache that belong to this amp.
                                 */
                                if (svd->type == MAP_SHARED) {
                                        ASSERT(amp->refcnt == 1);
                                        ASSERT(svd->softlockcnt == 0);
                                        anonmap_purge(amp);
                                }

                                if (amp->a_szc == seg->s_szc) {
                                        if (seg->s_szc != 0) {
                                                anon_free_pages(amp->ahp,
                                                    an_idx, len,
                                                    seg->s_szc);
                                        } else {
                                                anon_free(amp->ahp, an_idx,
                                                    len);
                                        }
                                } else {
                                        ASSERT(svd->type == MAP_SHARED);
                                        ASSERT(amp->a_szc > seg->s_szc);
                                        anon_shmap_free_pages(amp,
                                            an_idx, len);
                                }

                                /*
                                 * Unreserve swap space for the
                                 * unmapped chunk of this segment in
                                 * case it's MAP_SHARED
                                 */
                                if (svd->type == MAP_SHARED) {
                                        anon_unresv_zone(len,
                                            seg->s_as->a_proc->p_zone);
                                        amp->swresv -= len;
                                }
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }

                seg->s_size -= len;

                if (svd->swresv) {
                        if (svd->flags & MAP_NORESERVE) {
                                ASSERT(amp);
                                oswresv = svd->swresv;
                                svd->swresv = ptob(anon_pages(amp->ahp,
                                    svd->anon_index, npages));
                                anon_unresv_zone(oswresv - svd->swresv,
                                    seg->s_as->a_proc->p_zone);
                                if (SEG_IS_PARTIAL_RESV(seg))
                                        seg->s_as->a_resvsize -= oswresv -
                                            svd->swresv;
                        } else {
                                size_t unlen;
```

```
                                if (svd->pageswap) {
                                        oswresv = svd->swresv;
                                        svd->swresv =
                                            segvn_count_swap_by_vpages(seg);
                                        ASSERT(oswresv >= svd->swresv);
                                        unlen = oswresv - svd->swresv;
                                } else {
                                        svd->swresv -= len;
                                        ASSERT(svd->swresv == seg->s_size);
                                        unlen = len;
                                }
                                anon_unresv_zone(unlen,
                                    seg->s_as->a_proc->p_zone);
                        }
                        TRACE_3(TR_FAC_VM, TR_ANON_PROC,
                            "anon proc:%p %lu %u", seg, len, 0);
                }

                return (0);
        }

        /*
         * The section to go is in the middle of the segment,
         * have to make it into two segments.  nseg is made for
         * the high end while seg is cut down at the low end.
         */
        nbase = addr + len;                             /* new seg base */
        nsize = (seg->s_base + seg->s_size) - nbase;    /* new seg size */
        seg->s_size = addr - seg->s_base;               /* shrink old seg */
        nseg = seg_alloc(seg->s_as, nbase, nsize);
        if (nseg == NULL) {
                panic("segvn_unmap seg_alloc");
                /*NOTREACHED*/
        }
        nseg->s_ops = seg->s_ops;
        nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
        nseg->s_data = (void *)nsvd;
        nseg->s_szc = seg->s_szc;
        *nsvd = *svd;
        nsvd->seg = nseg;
        nsvd->offset = svd->offset + (uintptr_t)(nseg->s_base - seg->s_base);
        nsvd->swresv = 0;
        nsvd->softlockcnt = 0;
        nsvd->softlockcnt_sbase = 0;
        nsvd->softlockcnt_send = 0;
        ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);

        if (svd->vp != NULL) {
                VN_HOLD(nsvd->vp);
                if (nsvd->type == MAP_SHARED)
                        lgrp_shm_policy_init(NULL, nsvd->vp);
        }
        crhold(svd->cred);

        if (svd->vpage == NULL) {
                nsvd->vpage = NULL;
        } else {
                /* need to split vpage into two arrays */
                size_t nbytes;
                struct vpage *ovpage;

                ovpage = svd->vpage;            /* keep pointer to vpage */

                npages = seg_pages(seg);        /* seg has shrunk */
                nbytes = vpgtob(npages);
                svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
                bcopy(ovpage, svd->vpage, nbytes);

                npages = seg_pages(nseg);
                nbytes = vpgtob(npages);
                nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);

                bcopy(&ovpage[opages - npages], nsvd->vpage, nbytes);

                /* free up old vpage */
                kmem_free(ovpage, vpgtob(opages));
        }

        if (amp == NULL) {
                nsvd->amp = NULL;
                nsvd->anon_index = 0;
        } else {
                /*
                 * Need to create a new anon map for the new segment.
                 * We’ll also allocate a new smaller array for the old
                 * smaller segment to save space.
                 */
                opages = btop((uintptr_t)(addr - seg->s_base));
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
                        /*
                         * Free up now unused parts of anon_map array.
                         */
                        ulong_t an_idx = svd->anon_index + opages;

                        /*
                         * Shared anon map is no longer in use. Before
                         * freeing its pages purge all entries from
                         * pcache that belong to this amp.
                         */
                        if (svd->type == MAP_SHARED) {
                                ASSERT(amp->refcnt == 1);
                                ASSERT(svd->softlockcnt == 0);
                                anonmap_purge(amp);
                        }

                        if (amp->a_szc == seg->s_szc) {
                                if (seg->s_szc != 0) {
                                        anon_free_pages(amp->ahp, an_idx, len,
                                            seg->s_szc);
                                } else {
                                        anon_free(amp->ahp, an_idx,
                                            len);
                                }
                        } else {
                                ASSERT(svd->type == MAP_SHARED);
                                ASSERT(amp->a_szc > seg->s_szc);
                                anon_shmap_free_pages(amp, an_idx, len);
                        }

                        /*
                         * Unreserve swap space for the
                         * unmapped chunk of this segment in
                         * case it’s MAP_SHARED
                         */
                        if (svd->type == MAP_SHARED) {
                                anon_unresv_zone(len,
                                    seg->s_as->a_proc->p_zone);
                                amp->swresv -= len;
                        }
                }
```

```
                nsvd->anon_index = svd->anon_index +
                    btop((uintptr_t)(nseg->s_base - seg->s_base));
                if (svd->type == MAP_SHARED) {
                        amp->refcnt++;
                        nsvd->amp = amp;
                } else {
                        struct anon_map *namp;
                        struct anon_hdr *nahp;

                        ASSERT(svd->type == MAP_PRIVATE);
                        nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
                        namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
                        namp->a_szc = seg->s_szc;
                        (void) anon_copy_ptr(amp->ahp, svd->anon_index, nahp,
                            0, btop(seg->s_size), ANON_SLEEP);
                        (void) anon_copy_ptr(amp->ahp, nsvd->anon_index,
                            namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
                        anon_release(amp->ahp, btop(amp->size));
                        svd->anon_index = 0;
                        nsvd->anon_index = 0;
                        amp->ahp = nahp;
                        amp->size = seg->s_size;
                        nsvd->amp = namp;
                }
                ANON_LOCK_EXIT(&amp->a_rwlock);
        }
        if (svd->swresv) {
                if (svd->flags & MAP_NORESERVE) {
                        ASSERT(amp);
                        oswresv = svd->swresv;
                        svd->swresv = ptob(anon_pages(amp->ahp,
                            svd->anon_index, btop(seg->s_size)));
                        nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
                            nsvd->anon_index, btop(nseg->s_size)));
                        ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
                        anon_unresv_zone(oswresv - (svd->swresv + nsvd->swresv),
                            seg->s_as->a_proc->p_zone);
                        if (SEG_IS_PARTIAL_RESV(seg))
                                seg->s_as->a_resvsize -= oswresv -
                                    (svd->swresv + nsvd->swresv);
                } else {
                        size_t unlen;

                        if (svd->pageswap) {
                                oswresv = svd->swresv;
                                svd->swresv = segvn_count_swap_by_vpages(seg);
                                nsvd->swresv = segvn_count_swap_by_vpages(nseg);
                                ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
                                unlen = oswresv - (svd->swresv + nsvd->swresv);
                        } else {
                                if (seg->s_size + nseg->s_size + len !=
                                    svd->swresv) {
                                        panic("segvn_unmap: cannot split "
                                            "swap reservation");
                                        /*NOTREACHED*/
                                }
                                svd->swresv = seg->s_size;
                                nsvd->swresv = nseg->s_size;
                                unlen = len;
                        }
                        anon_unresv_zone(unlen,
                            seg->s_as->a_proc->p_zone);
                }
                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
                    seg, len, 0);
        }
        return (0);                     /* I’m glad that’s all over with! */
}

static void
segvn_free(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        pgcnt_t npages = seg_pages(seg);
        struct anon_map *amp;
        size_t len;

        /*
         * We don’t need any segment level locks for "segvn" data
         * since the address space is "write" locked.
         */
        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(svd->tr_state == SEGVN_TR_OFF);

        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);

        /*
         * Be sure to unlock pages. XXX Why do things get free’ed instead
         * of unmapped? XXX
         */
        (void) segvn_lockop(seg, seg->s_base, seg->s_size,
            0, MC_UNLOCK, NULL, 0);

        /*
         * Deallocate the vpage and anon pointers if necessary and possible.
         */
        if (svd->vpage != NULL) {
                kmem_free(svd->vpage, vpgtob(npages));
                svd->vpage = NULL;
        }
        if ((amp = svd->amp) != NULL) {
                /*
                 * If there are no more references to this anon_map
                 * structure, then deallocate the structure after freeing
                 * up all the anon slot pointers that we can.
                 */
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                ASSERT(amp->a_szc >= seg->s_szc);
                if (--amp->refcnt == 0) {
                        if (svd->type == MAP_PRIVATE) {
                                /*
                                 * Private - we only need to anon_free
                                 * the part that this segment refers to.
                                 */
                                if (seg->s_szc != 0) {
                                        anon_free_pages(amp->ahp,
                                            svd->anon_index, seg->s_size,
                                            seg->s_szc);
                                } else {
                                        anon_free(amp->ahp, svd->anon_index,
                                            seg->s_size);
                                }
                        } else {

                                /*
                                 * Shared anon map is no longer in use. Before
                                 * freeing its pages purge all entries from
                                 * pcache that belong to this amp.
                                 */
                                ASSERT(svd->softlockcnt == 0);
                                anonmap_purge(amp);
```

```
                                /*
                                 * Shared - anon_free the entire
                                 * anon_map’s worth of stuff and
                                 * release any swap reservation.
                                 */
                                if (amp->a_szc != 0) {
                                        anon_shmap_free_pages(amp, 0,
                                            amp->size);
                                } else {
                                        anon_free(amp->ahp, 0, amp->size);
                                }
                                if ((len = amp->swresv) != 0) {
                                        anon_unresv_zone(len,
                                            seg->s_as->a_proc->p_zone);
                                        TRACE_3(TR_FAC_VM, TR_ANON_PROC,
                                            "anon proc:%p %lu %u", seg, len, 0);
                                }
                        }
                        svd->amp = NULL;
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        anonmap_free(amp);
                } else if (svd->type == MAP_PRIVATE) {
                        /*
                         * We had a private mapping which still has
                         * a held anon_map so just free up all the
                         * anon slot pointers that we were using.
                         */
                        if (seg->s_szc != 0) {
                                anon_free_pages(amp->ahp, svd->anon_index,
                                    seg->s_size, seg->s_szc);
                        } else {
                                anon_free(amp->ahp, svd->anon_index,
                                    seg->s_size);
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }
        }

        /*
         * Release swap reservation.
         */
        if ((len = svd->swresv) != 0) {
                anon_unresv_zone(svd->swresv,
                    seg->s_as->a_proc->p_zone);
                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
                    seg, len, 0);
                if (SEG_IS_PARTIAL_RESV(seg))
                        seg->s_as->a_resvsize -= svd->swresv;
                svd->swresv = 0;
        }
        /*
         * Release claim on vnode, credentials, and finally free the
         * private data.
         */
        if (svd->vp != NULL) {
                if (svd->type == MAP_SHARED)
                        lgrp_shm_policy_fini(NULL, svd->vp);
                VN_RELE(svd->vp);
                svd->vp = NULL;
        }
        crfree(svd->cred);
        svd->pageprot = 0;
        svd->pageadvice = 0;
        svd->pageswap = 0;
        svd->cred = NULL;

        /*
         * Take segfree_syncmtx lock to let segvn_reclaim() finish if it's
         * still working with this segment without holding as lock (in case
         * it's called by pcache async thread).
         */
        ASSERT(svd->softlockcnt == 0);
        mutex_enter(&svd->segfree_syncmtx);
        mutex_exit(&svd->segfree_syncmtx);

        seg->s_data = NULL;
        kmem_cache_free(segvn_cache, svd);
}

/*
 * Do a F_SOFTUNLOCK call over the range requested.  The range must have
 * already been F_SOFTLOCK'ed.
 * Caller must always match addr and len of a softunlock with a previous
 * softlock with exactly the same addr and len.
 */
static void
segvn_softunlock(struct seg *seg, caddr_t addr, size_t len, enum seg_rw rw)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        page_t *pp;
        caddr_t adr;
        struct vnode *vp;
        u_offset_t offset;
        ulong_t anon_index;
        struct anon_map *amp;
        struct anon *ap = NULL;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

        if ((amp = svd->amp) != NULL)
                anon_index = svd->anon_index + seg_page(seg, addr);

        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                hat_unlock_region(seg->s_as->a_hat, addr, len, svd->rcookie);
        } else {
                hat_unlock(seg->s_as->a_hat, addr, len);
        }
        for (adr = addr; adr < addr + len; adr += PAGESIZE) {
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        if ((ap = anon_get_ptr(amp->ahp, anon_index++))
                            != NULL) {
                                swap_xlate(ap, &vp, &offset);
                        } else {
                                vp = svd->vp;
                                offset = svd->offset +
                                    (uintptr_t)(adr - seg->s_base);
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
                        vp = svd->vp;
                        offset = svd->offset +
                            (uintptr_t)(adr - seg->s_base);
                }

                /*
                 * Use page_find() instead of page_lookup() to
```

```
                 * find the page since we know that it is locked.
                 */
                pp = page_find(vp, offset);
                if (pp == NULL) {
                        panic(
                            "segvn_softunlock: addr %p, ap %p, vp %p, off %llx",
                            (void *)adr, (void *)ap, (void *)vp, offset);
                        /*NOTREACHED*/
                }

                if (rw == S_WRITE) {
                        hat_setrefmod(pp);
                        if (seg->s_as->a_vbits)
                                hat_setstat(seg->s_as, adr, PAGESIZE,
                                    P_REF | P_MOD);
                } else if (rw != S_OTHER) {
                        hat_setref(pp);
                        if (seg->s_as->a_vbits)
                                hat_setstat(seg->s_as, adr, PAGESIZE, P_REF);
                }
                TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
                    "segvn_fault:pp %p vp %p offset %llx", pp, vp, offset);
                page_unlock(pp);
        }
        ASSERT(svd->softlockcnt >= btop(len));
        if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -btop(len))) {
                /*
                 * All SOFTLOCKS are gone. Wakeup any waiting
                 * unmappers so they can try again to unmap.
                 * Check for waiters first without the mutex
                 * held so we don't always grab the mutex on
                 * softunlocks.
                 */
                if (AS_ISUNMAPWAIT(seg->s_as)) {
                        mutex_enter(&seg->s_as->a_contents);
                        if (AS_ISUNMAPWAIT(seg->s_as)) {
                                AS_CLRUNMAPWAIT(seg->s_as);
                                cv_broadcast(&seg->s_as->a_cv);
                        }
                        mutex_exit(&seg->s_as->a_contents);
                }
        }
}

#define PAGE_HANDLED    ((page_t *)-1)

/*
 * Release all the pages in the NULL terminated ppp list
 * which haven't already been converted to PAGE_HANDLED.
 */
static void
segvn_pagelist_rele(page_t **ppp)
{
        for (; *ppp != NULL; ppp++) {
                if (*ppp != PAGE_HANDLED)
                        page_unlock(*ppp);
        }
}

static int stealcow = 1;

/*
 * Workaround for viking chip bug.  See bug id 1220902.
 * To fix this down in pagefault() would require importing so
 * much as and segvn code as to be unmaintainable.
 */
```

```
int enable_mbit_wa = 0;

/*
 * Handles all the dirty work of getting the right
 * anonymous pages and loading up the translations.
 * This routine is called only from segvn_fault()
 * when looping over the range of addresses requested.
 *
 * The basic algorithm here is:
 *      If this is an anon_zero case
 *              Call anon_zero to allocate page
 *              Load up translation
 *              Return
 *      endif
 *      If this is an anon page
 *              Use anon_getpage to get the page
 *      else
 *              Find page in pl[] list passed in
 *      endif
 *      If not a cow
 *              Load up the translation to the page
 *              return
 *      endif
 *      Call anon_private to handle cow
 *      Load up (writable) translation to new page
 */
static faultcode_t
segvn_faultpage(
        struct hat *hat,                /* the hat to use for mapping */
        struct seg *seg,                /* seg_vn of interest */
        caddr_t addr,                   /* address in as */
        u_offset_t off,                 /* offset in vp */
        struct vpage *vpage,            /* pointer to vpage for vp, off */
        page_t *pl[],                   /* object source page pointer */
        uint_t vpprot,                  /* access allowed to object pages */
        enum fault_type type,           /* type of fault */
        enum seg_rw rw,                 /* type of access at fault */
        int brkcow)                     /* we may need to break cow */
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        page_t *pp, **ppp;
        uint_t pageflags = 0;
        page_t *anon_pl[1 + 1];
        page_t *opp = NULL;             /* original page */
        uint_t prot;
        int err;
        int cow;
        int claim;
        int steal = 0;
        ulong_t anon_index;
        struct anon *ap, *oldap;
        struct anon_map *amp;
        int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
        int anon_lock = 0;
        anon_sync_obj_t cookie;

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

        ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
        ASSERT(seg->s_szc == 0);
        ASSERT(svd->tr_state != SEGVN_TR_INIT);

        /*
         * Initialize protection value for this page.
```

```
         * If we have per page protection values check it now.
         */
        if (svd->pageprot) {
                uint_t protchk;

                switch (rw) {
                case S_READ:
                        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;
                }

                prot = VPP_PROT(vpage);
                if ((prot & protchk) == 0)
                        return (FC_PROT);       /* illegal access type */
        } else {
                prot = svd->prot;
        }

        if (type == F_SOFTLOCK) {
                atomic_inc_ulong((ulong_t *)&svd->softlockcnt);
        }

        /*
         * Always acquire the anon array lock to prevent 2 threads from
         * allocating separate anon slots for the same "addr".
         */

        if ((amp = svd->amp) != NULL) {
                ASSERT(RW_READ_HELD(&amp->a_rwlock));
                anon_index = svd->anon_index + seg_page(seg, addr);
                anon_array_enter(amp, anon_index, &cookie);
                anon_lock = 1;
        }

        if (svd->vp == NULL && amp != NULL) {
                if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL) {
                        /*
                         * Allocate a (normally) writable anonymous page of
                         * zeroes. If no advance reservations, reserve now.
                         */
                        if (svd->flags & MAP_NORESERVE) {
                                if (anon_resv_zone(ptob(1),
                                    seg->s_as->a_proc->p_zone)) {
                                        atomic_add_long(&svd->swresv, ptob(1));
                                        atomic_add_long(&seg->s_as->a_resvsize,
                                            ptob(1));
                                } else {
                                        err = ENOMEM;
                                        goto out;
                                }
                        }
                        if ((pp = anon_zero(seg, addr, &ap,
                            svd->cred)) == NULL) {
                                err = ENOMEM;
                                goto out;       /* out of swap space */
                        }
                        /*
                         * Re-acquire the anon_map lock and
                         * initialize the anon array entry.
                         */
                        (void) anon_set_ptr(amp->ahp, anon_index, ap,
                            ANON_SLEEP);

                        ASSERT(pp->p_szc == 0);

                        /*
                         * Handle pages that have been marked for migration
                         */
                        if (lgrp_optimizations())
                                page_migrate(seg, addr, &pp, 1);

                        if (enable_mbit_wa) {
                                if (rw == S_WRITE)
                                        hat_setmod(pp);
                                else if (!hat_ismod(pp))
                                        prot &= ~PROT_WRITE;
                        }
                        /*
                         * If AS_PAGLCK is set in a_flags (via memcntl(2)
                         * with MC_LOCKAS, MCL_FUTURE) and this is a
                         * MAP_NORESERVE segment, we may need to
                         * permanently lock the page as it is being faulted
                         * for the first time. The following text applies
                         * only to MAP_NORESERVE segments:
                         *
                         * As per memcntl(2), if this segment was created
                         * after MCL_FUTURE was applied (a "future"
                         * segment), its pages must be locked.  If this
                         * segment existed at MCL_FUTURE application (a
                         * "past" segment), the interface is unclear.
                         *
                         * We decide to lock only if vpage is present:
                         *
                         * - "future" segments will have a vpage array (see
                         *    as_map), and so will be locked as required
                         *
                         * - "past" segments may not have a vpage array,
                         *    depending on whether events (such as
                         *    mprotect) have occurred. Locking if vpage
                         *    exists will preserve legacy behavior.  Not
                         *    locking if vpage is absent, will not break
                         *    the interface or legacy behavior.  Note that
                         *    allocating vpage here if it’s absent requires
                         *    upgrading the segvn reader lock, the cost of
                         *    which does not seem worthwhile.
                         *
                         * Usually testing and setting VPP_ISPPLOCK and
                         * VPP_SETPPLOCK requires holding the segvn lock as
                         * writer, but in this case all readers are
                         * serializing on the anon array lock.
                         */
                        if (AS_ISPGLCK(seg->s_as) && vpage != NULL &&
                            (svd->flags & MAP_NORESERVE) &&
                            !VPP_ISPPLOCK(vpage)) {
                                proc_t *p = seg->s_as->a_proc;
                                ASSERT(svd->type == MAP_PRIVATE);
                                mutex_enter(&p->p_lock);
                                if (rctl_incr_locked_mem(p, NULL, PAGESIZE,
                                    1) == 0) {
                                        claim = VPP_PROT(vpage) & PROT_WRITE;
                                        if (page_pp_lock(pp, claim, 0)) {
                                                VPP_SETPPLOCK(vpage);
```

```
                                        } else {
                                                rctl_decr_locked_mem(p, NULL,
                                                    PAGESIZE, 1);
                                        }
                                }
                                mutex_exit(&p->p_lock);
                        }

                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                        hat_memload(hat, addr, pp, prot, hat_flag);

                        if (!(hat_flag & HAT_LOAD_LOCK))
                                page_unlock(pp);

                        anon_array_exit(&cookie);
                        return (0);
                }
        }

        /*
         * Obtain the page structure via anon_getpage() if it is
         * a private copy of an object (the result of a previous
         * copy-on-write).
         */
        if (amp != NULL) {
                if ((ap = anon_get_ptr(amp->ahp, anon_index)) != NULL) {
                        err = anon_getpage(&ap, &vpprot, anon_pl, PAGESIZE,
                            seg, addr, rw, svd->cred);
                        if (err)
                                goto out;

                        if (svd->type == MAP_SHARED) {
                                /*
                                 * If this is a shared mapping to an
                                 * anon_map, then ignore the write
                                 * permissions returned by anon_getpage().
                                 * They apply to the private mappings
                                 * of this anon_map.
                                 */
                                vpprot |= PROT_WRITE;
                        }
                        opp = anon_pl[0];
                }
        }

        /*
         * Search the pl[] list passed in if it is from the
         * original object (i.e., not a private copy).
         */
        if (opp == NULL) {
                /*
                 * Find original page.  We must be bringing it in
                 * from the list in pl[].
                 */
                for (ppp = pl; (opp = *ppp) != NULL; ppp++) {
                        if (opp == PAGE_HANDLED)
                                continue;
                        ASSERT(opp->p_vnode == svd->vp); /* XXX */
                        if (opp->p_offset == off)
                                break;
                }
                if (opp == NULL) {
                        panic("segvn_faultpage not found");
                        /*NOTREACHED*/
                }
                *ppp = PAGE_HANDLED;
        }

        ASSERT(PAGE_LOCKED(opp));

        TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
            "segvn_fault:pp %p vp %p offset %llx", opp, NULL, 0);

        /*
         * The fault is treated as a copy-on-write fault if a
         * write occurs on a private segment and the object
         * page (i.e., mapping) is write protected.  We assume
         * that fatal protection checks have already been made.
         */

        if (brkcow) {
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                cow = !(vpprot & PROT_WRITE);
        } else if (svd->tr_state == SEGVN_TR_ON) {
                /*
                 * If we are doing text replication COW on first touch.
                 */
                ASSERT(amp != NULL);
                ASSERT(svd->vp != NULL);
                ASSERT(rw != S_WRITE);
                cow = (ap == NULL);
        } else {
                cow = 0;
        }

        /*
         * If not a copy-on-write case load the translation
         * and return.
         */
        if (cow == 0) {

                /*
                 * Handle pages that have been marked for migration
                 */
                if (lgrp_optimizations())
                        page_migrate(seg, addr, &opp, 1);

                if (IS_VMODSORT(opp->p_vnode) || enable_mbit_wa) {
                        if (rw == S_WRITE)
                                hat_setmod(opp);
                        else if (rw != S_OTHER && !hat_ismod(opp))
                                prot &= ~PROT_WRITE;
                }

                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
                    (!svd->pageprot && svd->prot == (prot & vpprot)));
                ASSERT(amp == NULL ||
                    svd->rcookie == HAT_INVALID_REGION_COOKIE);
                hat_memload_region(hat, addr, opp, prot & vpprot, hat_flag,
                    svd->rcookie);

                if (!(hat_flag & HAT_LOAD_LOCK))
                        page_unlock(opp);

                if (anon_lock) {
                        anon_array_exit(&cookie);
                }
                return (0);
        }

        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
```

```
        hat_setref(opp);

        ASSERT(amp != NULL && anon_lock);

        /*
         * Steal the page only if it isn’t a private page
         * since stealing a private page is not worth the effort.
         */
        if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL)
                steal = 1;

        /*
         * Steal the original page if the following conditions are true:
         *
         * We are low on memory, the page is not private, page is not large,
         * not shared, not modified, not ‘locked’ or if we have it ‘locked’
         * (i.e., p_cowcnt == 1 and p_lckcnt == 0, which also implies
         * that the page is not shared) and if it doesn’t have any
         * translations. page_struct_lock isn’t needed to look at p_cowcnt
         * and p_lckcnt because we first get exclusive lock on page.
         */
        (void) hat_pagesync(opp, HAT_SYNC_DONTZERO | HAT_SYNC_STOPON_MOD);

        if (stealcow && freemem < minfree && steal && opp->p_szc == 0 &&
            page_tryupgrade(opp) && !hat_ismod(opp) &&
            ((opp->p_lckcnt == 0 && opp->p_cowcnt == 0) ||
            (opp->p_lckcnt == 0 && opp->p_cowcnt == 1 &&
            vpage != NULL && VPP_ISPPLOCK(vpage)))) {
                /*
                 * Check if this page has other translations
                 * after unloading our translation.
                 */
                if (hat_page_is_mapped(opp)) {
                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                        hat_unload(seg->s_as->a_hat, addr, PAGESIZE,
                            HAT_UNLOAD);
                }

                /*
                 * hat_unload() might sync back someone else’s recent
                 * modification, so check again.
                 */
                if (!hat_ismod(opp) && !hat_page_is_mapped(opp))
                        pageflags |= STEAL_PAGE;
        }

        /*
         * If we have a vpage pointer, see if it indicates that we have
         * ‘‘locked’’ the page we map -- if so, tell anon_private to
         * transfer the locking resource to the new page.
         *
         * See Statement at the beginning of segvn_lockop regarding
         * the way lockcnts/cowcnts are handled during COW.
         *
         */
        if (vpage != NULL && VPP_ISPPLOCK(vpage))
                pageflags |= LOCK_PAGE;

        /*
         * Allocate a private page and perform the copy.
         * For MAP_NORESERVE reserve swap space now, unless this
         * is a cow fault on an existing anon page in which case
         * MAP_NORESERVE will have made advance reservations.
         */
        if ((svd->flags & MAP_NORESERVE) && (ap == NULL)) {
                if (anon_resv_zone(ptob(1), seg->s_as->a_proc->p_zone)) {
                        atomic_add_long(&svd->swresv, ptob(1));
                        atomic_add_long(&seg->s_as->a_resvsize, ptob(1));
                } else {
                        page_unlock(opp);
                        err = ENOMEM;
                        goto out;
                }
        }
        oldap = ap;
        pp = anon_private(&ap, seg, addr, prot, opp, pageflags, svd->cred);
        if (pp == NULL) {
                err = ENOMEM;   /* out of swap space */
                goto out;
        }

        /*
         * If we copied away from an anonymous page, then
         * we are one step closer to freeing up an anon slot.
         *
         * NOTE:  The original anon slot must be released while
         * holding the "anon_map" lock.  This is necessary to prevent
         * other threads from obtaining a pointer to the anon slot
         * which may be freed if its "refcnt" is 1.
         */
        if (oldap != NULL)
                anon_decref(oldap);

        (void) anon_set_ptr(amp->ahp, anon_index, ap, ANON_SLEEP);

        /*
         * Handle pages that have been marked for migration
         */
        if (lgrp_optimizations())
                page_migrate(seg, addr, &pp, 1);

        ASSERT(pp->p_szc == 0);

        ASSERT(!IS_VMODSORT(pp->p_vnode));
        if (enable_mbit_wa) {
                if (rw == S_WRITE)
                        hat_setmod(pp);
                else if (!hat_ismod(pp))
                        prot &= ~PROT_WRITE;
        }

        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
        hat_memload(hat, addr, pp, prot, hat_flag);

        if (!(hat_flag & HAT_LOAD_LOCK))
                page_unlock(pp);

        ASSERT(anon_lock);
        anon_array_exit(&cookie);
        return (0);
out:
        if (anon_lock)
                anon_array_exit(&cookie);

        if (type == F_SOFTLOCK) {
                atomic_dec_ulong((ulong_t *)&svd->softlockcnt);
        }
        return (FC_MAKE_ERR(err));
}

/*
```

```
 * relocate a bunch of smaller targ pages into one large repl page. all targ
 * pages must be complete pages smaller than replacement pages.
 * it’s assumed that no page’s szc can change since they are all PAGESIZE or
 * complete large pages locked SHARED.
 */
static void
segvn_relocate_pages(page_t **targ, page_t *replacement)
{
        page_t *pp;
        pgcnt_t repl_npgs, curnpgs;
        pgcnt_t i;
        uint_t repl_szc = replacement->p_szc;
        page_t *first_repl = replacement;
        page_t *repl;
        spgcnt_t npgs;

        VM_STAT_ADD(segvnvmstats.relocatepages[0]);

        ASSERT(repl_szc != 0);
        npgs = repl_npgs = page_get_pagecnt(repl_szc);

        i = 0;
        while (repl_npgs) {
                spgcnt_t nreloc;
                int err;
                ASSERT(replacement != NULL);
                pp = targ[i];
                ASSERT(pp->p_szc < repl_szc);
                ASSERT(PAGE_EXCL(pp));
                ASSERT(!PP_ISFREE(pp));
                curnpgs = page_get_pagecnt(pp->p_szc);
                if (curnpgs == 1) {
                        VM_STAT_ADD(segvnvmstats.relocatepages[1]);
                        repl = replacement;
                        page_sub(&replacement, repl);
                        ASSERT(PAGE_EXCL(repl));
                        ASSERT(!PP_ISFREE(repl));
                        ASSERT(repl->p_szc == repl_szc);
                } else {
                        page_t *repl_savepp;
                        int j;
                        VM_STAT_ADD(segvnvmstats.relocatepages[2]);
                        repl_savepp = replacement;
                        for (j = 0; j < curnpgs; j++) {
                                repl = replacement;
                                page_sub(&replacement, repl);
                                ASSERT(PAGE_EXCL(repl));
                                ASSERT(!PP_ISFREE(repl));
                                ASSERT(repl->p_szc == repl_szc);
                                ASSERT(page_pptonum(targ[i + j]) ==
                                    page_pptonum(targ[i]) + j);
                        }
                        repl = repl_savepp;
                        ASSERT(IS_P2ALIGNED(page_pptonum(repl), curnpgs));
                }
                err = page_relocate(&pp, &repl, 0, 1, &nreloc, NULL);
                if (err || nreloc != curnpgs) {
                        panic("segvn_relocate_pages: "
                            "page_relocate failed err=%d curnpgs=%ld "
                            "nreloc=%ld", err, curnpgs, nreloc);
                }
                ASSERT(curnpgs <= repl_npgs);
                repl_npgs -= curnpgs;
                i += curnpgs;
        }
        ASSERT(replacement == NULL);
        repl = first_repl;
        repl_npgs = npgs;
        for (i = 0; i < repl_npgs; i++) {
                ASSERT(PAGE_EXCL(repl));
                ASSERT(!PP_ISFREE(repl));
                targ[i] = repl;
                page_downgrade(targ[i]);
                repl++;
        }
}

/*
 * Check if all pages in ppa array are complete smaller than szc pages and
 * their roots will still be aligned relative to their current size if the
 * entire ppa array is relocated into one szc page. If these conditions are
 * not met return 0.
 *
 * If all pages are properly aligned attempt to upgrade their locks
 * to exclusive mode. If it fails set *upgrdfail to 1 and return 0.
 * upgrdfail was set to 0 by caller.
 *
 * Return 1 if all pages are aligned and locked exclusively.
 *
 * If all pages in ppa array happen to be physically contiguous to make one
 * szc page and all exclusive locks are successfully obtained promote the page
 * size to szc and set *pszc to szc. Return 1 with pages locked shared.
 */
static int
segvn_full_szcpages(page_t **ppa, uint_t szc, int *upgrdfail, uint_t *pszc)
{
        page_t *pp;
        pfn_t pfn;
        pgcnt_t totnpgs = page_get_pagecnt(szc);
        pfn_t first_pfn;
        int contig = 1;
        pgcnt_t i;
        pgcnt_t j;
        uint_t curszc;
        pgcnt_t curnpgs;
        int root = 0;

        ASSERT(szc > 0);

        VM_STAT_ADD(segvnvmstats.fullszcpages[0]);

        for (i = 0; i < totnpgs; i++) {
                pp = ppa[i];
                ASSERT(PAGE_SHARED(pp));
                ASSERT(!PP_ISFREE(pp));
                pfn = page_pptonum(pp);
                if (i == 0) {
                        if (!IS_P2ALIGNED(pfn, totnpgs)) {
                                contig = 0;
                        } else {
                                first_pfn = pfn;
                        }
                } else if (contig && pfn != first_pfn + i) {
                        contig = 0;
                }
                if (pp->p_szc == 0) {
                        if (root) {
                                VM_STAT_ADD(segvnvmstats.fullszcpages[1]);
                                return (0);
                        }
                } else if (!root) {
```

```
                        if ((curszc = pp->p_szc) >= szc) {
                                VM_STAT_ADD(segvnvmstats.fullszcpages[2]);
                                return (0);
                        }
                        if (curszc == 0) {
                                /*
                                 * p_szc changed means we don’t have all pages
                                 * locked. return failure.
                                 */
                                VM_STAT_ADD(segvnvmstats.fullszcpages[3]);
                                return (0);
                        }
                        curnpgs = page_get_pagecnt(curszc);
                        if (!IS_P2ALIGNED(pfn, curnpgs) ||
                            !IS_P2ALIGNED(i, curnpgs)) {
                                VM_STAT_ADD(segvnvmstats.fullszcpages[4]);
                                return (0);
                        }
                        root = 1;
                } else {
                        ASSERT(i > 0);
                        VM_STAT_ADD(segvnvmstats.fullszcpages[5]);
                        if (pp->p_szc != curszc) {
                                VM_STAT_ADD(segvnvmstats.fullszcpages[6]);
                                return (0);
                        }
                        if (pfn - 1 != page_pptonum(ppa[i - 1])) {
                                panic("segvn_full_szcpages: "
                                    "large page not physically contiguous");
                        }
                        if (P2PHASE(pfn, curnpgs) == curnpgs - 1) {
                                root = 0;
                        }
                }
        }

        for (i = 0; i < totnpgs; i++) {
                ASSERT(ppa[i]->p_szc < szc);
                if (!page_tryupgrade(ppa[i])) {
                        for (j = 0; j < i; j++) {
                                page_downgrade(ppa[j]);
                        }
                        *pszc = ppa[i]->p_szc;
                        *upgrdfail = 1;
                        VM_STAT_ADD(segvnvmstats.fullszcpages[7]);
                        return (0);
                }
        }

        /*
         * When a page is put a free cachelist its szc is set to 0.  if file
         * system reclaimed pages from cachelist targ pages will be physically
         * contiguous with 0 p_szc.  in this case just upgrade szc of targ
         * pages without any relocations.
         * To avoid any hat issues with previous small mappings
         * hat_pageunload() the target pages first.
         */
        if (contig) {
                VM_STAT_ADD(segvnvmstats.fullszcpages[8]);
                for (i = 0; i < totnpgs; i++) {
                        (void) hat_pageunload(ppa[i], HAT_FORCE_PGUNLOAD);
                }
                for (i = 0; i < totnpgs; i++) {
                        ppa[i]->p_szc = szc;
                }
                for (i = 0; i < totnpgs; i++) {
                        ASSERT(PAGE_EXCL(ppa[i]));
                        page_downgrade(ppa[i]);
                }
                if (pszc != NULL) {
                        *pszc = szc;
                }
        }
        VM_STAT_ADD(segvnvmstats.fullszcpages[9]);
        return (1);
}

/*
 * Create physically contiguous pages for [vp, off] - [vp, off +
 * page_size(szc)) range and for private segment return them in ppa array.
 * Pages are created either via IO or relocations.
 *
 * Return 1 on success and 0 on failure.
 *
 * If physically contiguous pages already exist for this range return 1 without
 * filling ppa array. Caller initializes ppa[0] as NULL to detect that ppa
 * array wasn’t filled. In this case caller fills ppa array via VOP_GETPAGE().
 */

static int
segvn_fill_vp_pages(struct segvn_data *svd, vnode_t *vp, u_offset_t off,
    uint_t szc, page_t **ppa, page_t **ppplist, uint_t *ret_pszc,
    int *downsize)

{
        page_t *pplist = *ppplist;
        size_t pgsz = page_get_pagesize(szc);
        pgcnt_t pages = btop(pgsz);
        ulong_t start_off = off;
        u_offset_t eoff = off + pgsz;
        spgcnt_t nreloc;
        u_offset_t io_off = off;
        size_t io_len;
        page_t *io_pplist = NULL;
        page_t *done_pplist = NULL;
        pgcnt_t pgidx = 0;
        page_t *pp;
        page_t *newpp;
        page_t *targpp;
        int io_err = 0;
        int i;
        pfn_t pfn;
        ulong_t ppages;
        page_t *targ_pplist = NULL;
        page_t *repl_pplist = NULL;
        page_t *tmp_pplist;
        int nios = 0;
        uint_t pszc;
        struct vattr va;

        VM_STAT_ADD(segvnvmstats.fill_vp_pages[0]);

        ASSERT(szc != 0);
        ASSERT(pplist->p_szc == szc);

        /*
         * downsize will be set to 1 only if we fail to lock pages. this will
         * allow subsequent faults to try to relocate the page again. If we
         * fail due to misalignment don’t downsize and let the caller map the
         * whole region with small mappings to avoid more faults into the area
         * where we can’t get large pages anyway.
         */
```

```
        *downsize = 0;

        while (off < eoff) {
                newpp = pplist;
                ASSERT(newpp != NULL);
                ASSERT(PAGE_EXCL(newpp));
                ASSERT(!PP_ISFREE(newpp));
                /*
                 * we pass NULL for nrelocp to page_lookup_create()
                 * so that it doesn’t relocate. We relocate here
                 * later only after we make sure we can lock all
                 * pages in the range we handle and they are all
                 * aligned.
                 */
                pp = page_lookup_create(vp, off, SE_SHARED, newpp, NULL, 0);
                ASSERT(pp != NULL);
                ASSERT(!PP_ISFREE(pp));
                ASSERT(pp->p_vnode == vp);
                ASSERT(pp->p_offset == off);
                if (pp == newpp) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[1]);
                        page_sub(&pplist, pp);
                        ASSERT(PAGE_EXCL(pp));
                        ASSERT(page_iolock_assert(pp));
                        page_list_concat(&io_pplist, &pp);
                        off += PAGESIZE;
                        continue;
                }
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[2]);
                pfn = page_pptonum(pp);
                pszc = pp->p_szc;
                if (pszc >= szc && targ_pplist == NULL && io_pplist == NULL &&
                    IS_P2ALIGNED(pfn, pages)) {
                        ASSERT(repl_pplist == NULL);
                        ASSERT(done_pplist == NULL);
                        ASSERT(pplist == *ppplist);
                        page_unlock(pp);
                        page_free_replacement_page(pplist);
                        page_create_putback(pages);
                        *ppplist = NULL;
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[3]);
                        return (1);
                }
                if (pszc >= szc) {
                        page_unlock(pp);
                        segvn_faultvnmpss_align_err1++;
                        goto out;
                }
                ppages = page_get_pagecnt(pszc);
                if (!IS_P2ALIGNED(pfn, ppages)) {
                        ASSERT(pszc > 0);
                        /*
                         * sizing down to pszc won’t help.
                         */
                        page_unlock(pp);
                        segvn_faultvnmpss_align_err2++;
                        goto out;
                }
                pfn = page_pptonum(newpp);
                if (!IS_P2ALIGNED(pfn, ppages)) {
                        ASSERT(pszc > 0);
                        /*
                         * sizing down to pszc won’t help.
                         */
                        page_unlock(pp);
                        segvn_faultvnmpss_align_err3++;
                        goto out;
                }
                if (!PAGE_EXCL(pp)) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[4]);
                        page_unlock(pp);
                        *downsize = 1;
                        *ret_pszc = pp->p_szc;
                        goto out;
                }
                targpp = pp;
                if (io_pplist != NULL) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[5]);
                        io_len = off - io_off;
                        /*
                         * Some file systems like NFS don't check EOF
                         * conditions in VOP_PAGEIO(). Check it here
                         * now that pages are locked SE_EXCL. Any file
                         * truncation will wait until the pages are
                         * unlocked so no need to worry that file will
                         * be truncated after we check its size here.
                         * XXX fix NFS to remove this check.
                         */
                        va.va_mask = AT_SIZE;
                        if (VOP_GETATTR(vp, &va, ATTR_HINT, svd->cred, NULL)) {
                                VM_STAT_ADD(segvnvmstats.fill_vp_pages[6]);
                                page_unlock(targpp);
                                goto out;
                        }
                        if (btopr(va.va_size) < btopr(io_off + io_len)) {
                                VM_STAT_ADD(segvnvmstats.fill_vp_pages[7]);
                                *downsize = 1;
                                *ret_pszc = 0;
                                page_unlock(targpp);
                                goto out;
                        }
                        io_err = VOP_PAGEIO(vp, io_pplist, io_off, io_len,
                                B_READ, svd->cred, NULL);
                        if (io_err) {
                                VM_STAT_ADD(segvnvmstats.fill_vp_pages[8]);
                                page_unlock(targpp);
                                if (io_err == EDEADLK) {
                                        segvn_vmpss_pageio_deadlk_err++;
                                }
                                goto out;
                        }
                        nios++;
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[9]);
                        while (io_pplist != NULL) {
                                pp = io_pplist;
                                page_sub(&io_pplist, pp);
                                ASSERT(page_iolock_assert(pp));
                                page_io_unlock(pp);
                                pgidx = (pp->p_offset - start_off) >>
                                    PAGESHIFT;
                                ASSERT(pgidx < pages);
                                ppa[pgidx] = pp;
                                page_list_concat(&done_pplist, &pp);
                        }
                }
                pp = targpp;
                ASSERT(PAGE_EXCL(pp));
                ASSERT(pp->p_szc <= pszc);
                if (pszc != 0 && !group_page_trylock(pp, SE_EXCL)) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[10]);
                        page_unlock(pp);
                        *downsize = 1;
```

```
                                *ret_pszc = pp->p_szc;
                                goto out;
                        }
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[11]);
                        /*
                         * page szc chould have changed before the entire group was
                         * locked. reread page szc.
                         */
                        pszc = pp->p_szc;
                        ppages = page_get_pagecnt(pszc);

                        /* link just the roots */
                        page_list_concat(&targ_pplist, &pp);
                        page_sub(&pplist, newpp);
                        page_list_concat(&repl_pplist, &newpp);
                        off += PAGESIZE;
                        while (--ppages != 0) {
                                newpp = pplist;
                                page_sub(&pplist, newpp);
                                off += PAGESIZE;
                        }
                        io_off = off;
        }
        if (io_pplist != NULL) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[12]);
                io_len = eoff - io_off;
                va.va_mask = AT_SIZE;
                if (VOP_GETATTR(vp, &va, ATTR_HINT, svd->cred, NULL) != 0) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[13]);
                        goto out;
                }
                if (btopr(va.va_size) < btopr(io_off + io_len)) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[14]);
                        *downsize = 1;
                        *ret_pszc = 0;
                        goto out;
                }
                io_err = VOP_PAGEIO(vp, io_pplist, io_off, io_len,
                    B_READ, svd->cred, NULL);
                if (io_err) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[15]);
                        if (io_err == EDEADLK) {
                                segvn_vmpss_pageio_deadlk_err++;
                        }
                        goto out;
                }
                nios++;
                while (io_pplist != NULL) {
                        pp = io_pplist;
                        page_sub(&io_pplist, pp);
                        ASSERT(page_iolock_assert(pp));
                        page_io_unlock(pp);
                        pgidx = (pp->p_offset - start_off) >> PAGESHIFT;
                        ASSERT(pgidx < pages);
                        ppa[pgidx] = pp;
                }
        }
        /*
         * we're now bound to succeed or panic.
         * remove pages from done_pplist. it's not needed anymore.
         */
        while (done_pplist != NULL) {
                pp = done_pplist;
                page_sub(&done_pplist, pp);
        }
        VM_STAT_ADD(segvnvmstats.fill_vp_pages[16]);
        ASSERT(pplist == NULL);
        *ppplist = NULL;
        while (targ_pplist != NULL) {
                int ret;
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[17]);
                ASSERT(repl_pplist);
                pp = targ_pplist;
                page_sub(&targ_pplist, pp);
                pgidx = (pp->p_offset - start_off) >> PAGESHIFT;
                newpp = repl_pplist;
                page_sub(&repl_pplist, newpp);
#ifdef DEBUG
                pfn = page_pptonum(pp);
                pszc = pp->p_szc;
                ppages = page_get_pagecnt(pszc);
                ASSERT(IS_P2ALIGNED(pfn, ppages));
                pfn = page_pptonum(newpp);
                ASSERT(IS_P2ALIGNED(pfn, ppages));
                ASSERT(P2PHASE(pfn, pages) == pgidx);
#endif
                nreloc = 0;
                ret = page_relocate(&pp, &newpp, 0, 1, &nreloc, NULL);
                if (ret != 0 || nreloc == 0) {
                        panic("segvn_fill_vp_pages: "
                            "page_relocate failed");
                }
                pp = newpp;
                while (nreloc-- != 0) {
                        ASSERT(PAGE_EXCL(pp));
                        ASSERT(pp->p_vnode == vp);
                        ASSERT(pgidx ==
                            ((pp->p_offset - start_off) >> PAGESHIFT));
                        ppa[pgidx++] = pp;
                        pp++;
                }
        }

        if (svd->type == MAP_PRIVATE) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[18]);
                for (i = 0; i < pages; i++) {
                        ASSERT(ppa[i] != NULL);
                        ASSERT(PAGE_EXCL(ppa[i]));
                        ASSERT(ppa[i]->p_vnode == vp);
                        ASSERT(ppa[i]->p_offset ==
                            start_off + (i << PAGESHIFT));
                        page_downgrade(ppa[i]);
                }
                ppa[pages] = NULL;
        } else {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[19]);
                /*
                 * the caller will still call VOP_GETPAGE() for shared segments
                 * to check FS write permissions. For private segments we map
                 * file read only anyway.  so no VOP_GETPAGE is needed.
                 */
                for (i = 0; i < pages; i++) {
                        ASSERT(ppa[i] != NULL);
                        ASSERT(PAGE_EXCL(ppa[i]));
                        ASSERT(ppa[i]->p_vnode == vp);
                        ASSERT(ppa[i]->p_offset ==
                            start_off + (i << PAGESHIFT));
                        page_unlock(ppa[i]);
                }
                ppa[0] = NULL;
        }
```

```
        return (1);
out:
        /*
         * Do the cleanup. Unlock target pages we didn’t relocate. They are
         * linked on targ_pplist by root pages. reassemble unused replacement
         * and io pages back to pplist.
         */
        if (io_pplist != NULL) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[20]);
                pp = io_pplist;
                do {
                        ASSERT(pp->p_vnode == vp);
                        ASSERT(pp->p_offset == io_off);
                        ASSERT(page_iolock_assert(pp));
                        page_io_unlock(pp);
                        page_hashout(pp, NULL);
                        io_off += PAGESIZE;
                } while ((pp = pp->p_next) != io_pplist);
                page_list_concat(&io_pplist, &pplist);
                pplist = io_pplist;
        }
        tmp_pplist = NULL;
        while (targ_pplist != NULL) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[21]);
                pp = targ_pplist;
                ASSERT(PAGE_EXCL(pp));
                page_sub(&targ_pplist, pp);

                pszc = pp->p_szc;
                ppages = page_get_pagecnt(pszc);
                ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));

                if (pszc != 0) {
                        group_page_unlock(pp);
                }
                page_unlock(pp);

                pp = repl_pplist;
                ASSERT(pp != NULL);
                ASSERT(PAGE_EXCL(pp));
                ASSERT(pp->p_szc == szc);
                page_sub(&repl_pplist, pp);

                ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));

                /* relink replacement page */
                page_list_concat(&tmp_pplist, &pp);
                while (--ppages != 0) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[22]);
                        pp++;
                        ASSERT(PAGE_EXCL(pp));
                        ASSERT(pp->p_szc == szc);
                        page_list_concat(&tmp_pplist, &pp);
                }
        }
        if (tmp_pplist != NULL) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[23]);
                page_list_concat(&tmp_pplist, &pplist);
                pplist = tmp_pplist;
        }
        /*
         * at this point all pages are either on done_pplist or
         * pplist. They can’t be all on done_pplist otherwise
         * we’d’ve been done.
         */
        ASSERT(pplist != NULL);
        if (nios != 0) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[24]);
                pp = pplist;
                do {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[25]);
                        ASSERT(pp->p_szc == szc);
                        ASSERT(PAGE_EXCL(pp));
                        ASSERT(pp->p_vnode != vp);
                        pp->p_szc = 0;
                } while ((pp = pp->p_next) != pplist);

                pp = done_pplist;
                do {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[26]);
                        ASSERT(pp->p_szc == szc);
                        ASSERT(PAGE_EXCL(pp));
                        ASSERT(pp->p_vnode == vp);
                        pp->p_szc = 0;
                } while ((pp = pp->p_next) != done_pplist);

                while (pplist != NULL) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[27]);
                        pp = pplist;
                        page_sub(&pplist, pp);
                        page_free(pp, 0);
                }

                while (done_pplist != NULL) {
                        VM_STAT_ADD(segvnvmstats.fill_vp_pages[28]);
                        pp = done_pplist;
                        page_sub(&done_pplist, pp);
                        page_unlock(pp);
                }
                *ppplist = NULL;
                return (0);
        }
        ASSERT(pplist == *ppplist);
        if (io_err) {
                VM_STAT_ADD(segvnvmstats.fill_vp_pages[29]);
                /*
                 * don’t downsize on io error.
                 * see if vop_getpage succeeds.
                 * pplist may still be used in this case
                 * for relocations.
                 */
                return (0);
        }
        VM_STAT_ADD(segvnvmstats.fill_vp_pages[30]);
        page_free_replacement_page(pplist);
        page_create_putback(pages);
        *ppplist = NULL;
        return (0);
}

int segvn_anypgsz = 0;

#define SEGVN_RESTORE_SOFTLOCK_VP(type, pages)                          \
                if ((type) == F_SOFTLOCK) {                             \
                        atomic_add_long((ulong_t *)&(svd)->softlockcnt, \
                            -(pages));                                  \
                }

#define SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot)              \
                if (IS_VMODSORT((ppa)[0]->p_vnode)) {                   \
                        if ((rw) == S_WRITE) {                          \
                                for (i = 0; i < (pages); i++) {         \
                                        ASSERT((ppa)[i]->p_vnode ==     \
                                            (ppa)[0]->p_vnode);         \
                                        hat_setmod((ppa)[i]);           \
                                }                                       \
                        } else if ((rw) != S_OTHER &&                   \
                            ((prot) & (vpprot) & PROT_WRITE)) {         \
                                for (i = 0; i < (pages); i++) {         \
                                        ASSERT((ppa)[i]->p_vnode ==     \
                                            (ppa)[0]->p_vnode);         \
                                        if (!hat_ismod((ppa)[i])) {     \
                                                prot &= ~PROT_WRITE;    \
                                                break;                  \
                                        }                               \
                                }                                       \
                        }                                               \
                }

#ifdef  VM_STATS

#define SEGVN_VMSTAT_FLTVNPAGES(idx)                                    \
                VM_STAT_ADD(segvnvmstats.fltvnpages[(idx)]);

#else /* VM_STATS */

#define SEGVN_VMSTAT_FLTVNPAGES(idx)

#endif

static faultcode_t
segvn_fault_vnodepages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
    caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
    caddr_t eaddr, int brkcow)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp = svd->amp;
        uchar_t segtype = svd->type;
        uint_t szc = seg->s_szc;
        size_t pgsz = page_get_pagesize(szc);
        size_t maxpgsz = pgsz;
        pgcnt_t pages = btop(pgsz);
        pgcnt_t maxpages = pages;
        size_t ppasize = (pages + 1) * sizeof (page_t *);
        caddr_t a = lpgaddr;
        caddr_t maxlpgeaddr = lpgeaddr;
        u_offset_t off = svd->offset + (uintptr_t)(a - seg->s_base);
        ulong_t aindx = svd->anon_index + seg_page(seg, a);
        struct vpage *vpage = (svd->vpage != NULL) ?
            &svd->vpage[seg_page(seg, a)] : NULL;
        vnode_t *vp = svd->vp;
        page_t **ppa;
        uint_t  pszc;
        size_t  ppgsz;
        pgcnt_t ppages;
        faultcode_t err = 0;
        int ierr;
        int vop_size_err = 0;
        uint_t protchk, prot, vpprot;
        ulong_t i;
        int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
        anon_sync_obj_t an_cookie;
        enum seg_rw arw;
        int alloc_failed = 0;
        int adjszc_chk;
        struct vattr va;
        int xhat = 0;
        page_t *pplist;
        pfn_t pfn;
        int physcontig;
        int upgrdfail;
        int segvn_anypgsz_vnode = 0; /* for now map vnode with 2 page sizes */
        int tron = (svd->tr_state == SEGVN_TR_ON);

        ASSERT(szc != 0);
        ASSERT(vp != NULL);
        ASSERT(brkcow == 0 || amp != NULL);
        ASSERT(tron == 0 || amp != NULL);
        ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
        ASSERT(!(svd->flags & MAP_NORESERVE));
        ASSERT(type != F_SOFTUNLOCK);
        ASSERT(IS_P2ALIGNED(a, maxpgsz));
        ASSERT(amp == NULL || IS_P2ALIGNED(aindx, maxpages));
        ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
        ASSERT(seg->s_szc < NBBY * sizeof (int));
        ASSERT(type != F_SOFTLOCK || lpgeaddr - a == maxpgsz);
        ASSERT(svd->tr_state != SEGVN_TR_INIT);

        VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnvmstats.fltvnpages[0]);
        VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnvmstats.fltvnpages[1]);

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

        if (svd->pageprot) {
                switch (rw) {
                case S_READ:
                        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;
                }
        } else {
                prot = svd->prot;
                /* caller has already done segment level protection check. */
        }

        if (seg->s_as->a_hat != hat) {
                xhat = 1;
        }

        if (rw == S_WRITE && segtype == MAP_PRIVATE) {
                SEGVN_VMSTAT_FLTVNPAGES(2);
                arw = S_READ;
        } else {
                arw = rw;
        }

        ppa = kmem_alloc(ppasize, KM_SLEEP);

        VM_STAT_COND_ADD(amp != NULL, segvnvmstats.fltvnpages[3]);

        for (;;) {
                adjszc_chk = 0;
                for (; a < lpgeaddr; a += pgsz, off += pgsz, aindx += pages) {
```

```
                        if (adjszc_chk) {
                                while (szc < seg->s_szc) {
                                        uintptr_t e;
                                        uint_t tszc;
                                        tszc = segvn_anypgsz_vnode ? szc + 1 :
                                            seg->s_szc;
                                        ppgsz = page_get_pagesize(tszc);
                                        if (!IS_P2ALIGNED(a, ppgsz) ||
                                            ((alloc_failed >> tszc) & 0x1)) {
                                                break;
                                        }
                                        SEGVN_VMSTAT_FLTVNPAGES(4);
                                        szc = tszc;
                                        pgsz = ppgsz;
                                        pages = btop(pgsz);
                                        e = P2ROUNDUP((uintptr_t)eaddr, pgsz);
                                        lpgeaddr = (caddr_t)e;
                                }
                        }

                again:
                        if (IS_P2ALIGNED(a, maxpgsz) && amp != NULL) {
                                ASSERT(IS_P2ALIGNED(aindx, maxpages));
                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                                anon_array_enter(amp, aindx, &an_cookie);
                                if (anon_get_ptr(amp->ahp, aindx) != NULL) {
                                        SEGVN_VMSTAT_FLTVNPAGES(5);
                                        ASSERT(anon_pages(amp->ahp, aindx,
                                            maxpages) == maxpages);
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                        err = segvn_fault_anonpages(hat, seg,
                                            a, a + maxpgsz, type, rw,
                                            MAX(a, addr),
                                            MIN(a + maxpgsz, eaddr), brkcow);
                                        if (err != 0) {
                                                SEGVN_VMSTAT_FLTVNPAGES(6);
                                                goto out;
                                        }
                                        if (szc < seg->s_szc) {
                                                szc = seg->s_szc;
                                                pgsz = maxpgsz;
                                                pages = maxpages;
                                                lpgeaddr = maxlpgeaddr;
                                        }
                                        goto next;
                                } else {
                                        ASSERT(anon_pages(amp->ahp, aindx,
                                            maxpages) == 0);
                                        SEGVN_VMSTAT_FLTVNPAGES(7);
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                        }
                        ASSERT(!brkcow || IS_P2ALIGNED(a, maxpgsz));
                        ASSERT(!tron || IS_P2ALIGNED(a, maxpgsz));

                        if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
                                ASSERT(vpage != NULL);
                                prot = VPP_PROT(vpage);
                                ASSERT(sameprot(seg, a, maxpgsz));
                                if ((prot & protchk) == 0) {
                                        SEGVN_VMSTAT_FLTVNPAGES(8);
                                        err = FC_PROT;
                                        goto out;
                                }
                        }
                        if (type == F_SOFTLOCK) {
                                atomic_add_long((ulong_t *)&svd->softlockcnt,
                                    pages);
                        }

                        pplist = NULL;
                        physcontig = 0;
                        ppa[0] = NULL;
                        if (!brkcow && !tron && szc &&
                            !page_exists_physcontig(vp, off, szc,
                            segtype == MAP_PRIVATE ? ppa : NULL)) {
                                SEGVN_VMSTAT_FLTVNPAGES(9);
                                if (page_alloc_pages(vp, seg, a, &pplist, NULL,
                                    szc, 0, 0) && type != F_SOFTLOCK) {
                                        SEGVN_VMSTAT_FLTVNPAGES(10);
                                        pszc = 0;
                                        ierr = -1;
                                        alloc_failed |= (1 << szc);
                                        break;
                                }
                                if (pplist != NULL &&
                                    vp->v_mpssdata == SEGVN_PAGEIO) {
                                        int downsize;
                                        SEGVN_VMSTAT_FLTVNPAGES(11);
                                        physcontig = segvn_fill_vp_pages(svd,
                                            vp, off, szc, ppa, &pplist,
                                            &pszc, &downsize);
                                        ASSERT(!physcontig || pplist == NULL);
                                        if (!physcontig && downsize &&
                                            type != F_SOFTLOCK) {
                                                ASSERT(pplist == NULL);
                                                SEGVN_VMSTAT_FLTVNPAGES(12);
                                                ierr = -1;
                                                break;
                                        }
                                        ASSERT(!physcontig ||
                                            segtype == MAP_PRIVATE ||
                                            ppa[0] == NULL);
                                        if (physcontig && ppa[0] == NULL) {
                                                physcontig = 0;
                                        }
                                }
                        } else if (!brkcow && !tron && szc && ppa[0] != NULL) {
                                SEGVN_VMSTAT_FLTVNPAGES(13);
                                ASSERT(segtype == MAP_PRIVATE);
                                physcontig = 1;
                        }

                        if (!physcontig) {
                                SEGVN_VMSTAT_FLTVNPAGES(14);
                                ppa[0] = NULL;
                                ierr = VOP_GETPAGE(vp, (offset_t)off, pgsz,
                                    &vpprot, ppa, pgsz, seg, a, arw,
                                    svd->cred, NULL);
#ifdef DEBUG
                                if (ierr == 0) {
                                        for (i = 0; i < pages; i++) {
                                                ASSERT(PAGE_LOCKED(ppa[i]));
                                                ASSERT(!PP_ISFREE(ppa[i]));
                                                ASSERT(ppa[i]->p_vnode == vp);
                                                ASSERT(ppa[i]->p_offset ==
                                                    off + (i << PAGESHIFT));
                                        }
                                }
#endif /* DEBUG */
```

```
                                if (segtype == MAP_PRIVATE) {
                                        SEGVN_VMSTAT_FLTVNPAGES(15);
                                        vpprot &= ~PROT_WRITE;
                                }
                        } else {
                                ASSERT(segtype == MAP_PRIVATE);
                                SEGVN_VMSTAT_FLTVNPAGES(16);
                                vpprot = PROT_ALL & ~PROT_WRITE;
                                ierr = 0;
                        }

                        if (ierr != 0) {
                                SEGVN_VMSTAT_FLTVNPAGES(17);
                                if (pplist != NULL) {
                                        SEGVN_VMSTAT_FLTVNPAGES(18);
                                        page_free_replacement_page(pplist);
                                        page_create_putback(pages);
                                }
                                SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
                                if (a + pgsz <= eaddr) {
                                        SEGVN_VMSTAT_FLTVNPAGES(19);
                                        err = FC_MAKE_ERR(ierr);
                                        goto out;
                                }
                                va.va_mask = AT_SIZE;
                                if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL)) {
                                        SEGVN_VMSTAT_FLTVNPAGES(20);
                                        err = FC_MAKE_ERR(EIO);
                                        goto out;
                                }
                                if (btopr(va.va_size) >= btopr(off + pgsz)) {
                                        SEGVN_VMSTAT_FLTVNPAGES(21);
                                        err = FC_MAKE_ERR(ierr);
                                        goto out;
                                }
                                if (btopr(va.va_size) <
                                    btopr(off + (eaddr - a))) {
                                        SEGVN_VMSTAT_FLTVNPAGES(22);
                                        err = FC_MAKE_ERR(ierr);
                                        goto out;
                                }
                                if (brkcow || tron || type == F_SOFTLOCK) {
                                        /* can’t reduce map area */
                                        SEGVN_VMSTAT_FLTVNPAGES(23);
                                        vop_size_err = 1;
                                        goto out;
                                }
                                SEGVN_VMSTAT_FLTVNPAGES(24);
                                ASSERT(szc != 0);
                                pszc = 0;
                                ierr = -1;
                                break;
                        }

                        if (amp != NULL) {
                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                                anon_array_enter(amp, aindx, &an_cookie);
                        }
                        if (amp != NULL &&
                            anon_get_ptr(amp->ahp, aindx) != NULL) {
                                ulong_t taindx = P2ALIGN(aindx, maxpages);

                                SEGVN_VMSTAT_FLTVNPAGES(25);
                                ASSERT(anon_pages(amp->ahp, taindx,
                                    maxpages) == maxpages);
                                for (i = 0; i < pages; i++) {
                                        page_unlock(ppa[i]);
                                }
                                anon_array_exit(&an_cookie);
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                if (pplist != NULL) {
                                        page_free_replacement_page(pplist);
                                        page_create_putback(pages);
                                }
                                SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
                                if (szc < seg->s_szc) {
                                        SEGVN_VMSTAT_FLTVNPAGES(26);
                                        /*
                                         * For private segments SOFTLOCK
                                         * either always breaks cow (any rw
                                         * type except S_READ_NOCOW) or
                                         * address space is locked as writer
                                         * (S_READ_NOCOW case) and anon slots
                                         * can’t show up on second check.
                                         * Therefore if we are here for
                                         * SOFTLOCK case it must be a cow
                                         * break but cow break never reduces
                                         * szc. text replication (tron) in
                                         * this case works as cow break.
                                         * Thus the assert below.
                                         */
                                        ASSERT(!brkcow && !tron &&
                                            type != F_SOFTLOCK);
                                        pszc = seg->s_szc;
                                        ierr = -2;
                                        break;
                                }
                                ASSERT(IS_P2ALIGNED(a, maxpgsz));
                                goto again;
                        }
#ifdef DEBUG
                        if (amp != NULL) {
                                ulong_t taindx = P2ALIGN(aindx, maxpages);
                                ASSERT(!anon_pages(amp->ahp, taindx, maxpages));
                        }
#endif /* DEBUG */

                        if (brkcow || tron) {
                                ASSERT(amp != NULL);
                                ASSERT(pplist == NULL);
                                ASSERT(szc == seg->s_szc);
                                ASSERT(IS_P2ALIGNED(a, maxpgsz));
                                ASSERT(IS_P2ALIGNED(aindx, maxpages));
                                SEGVN_VMSTAT_FLTVNPAGES(27);
                                ierr = anon_map_privatepages(amp, aindx, szc,
                                    seg, a, prot, ppa, vpage, segvn_anypgsz,
                                    tron ? PG_LOCAL : 0, svd->cred);
                                if (ierr != 0) {
                                        SEGVN_VMSTAT_FLTVNPAGES(28);
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                        SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
                                        err = FC_MAKE_ERR(ierr);
                                        goto out;
                                }

                                ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));
                                /*
                                 * p_szc can’t be changed for locked
                                 * swapfs pages.
                                 */
                                ASSERT(svd->rcookie ==
```

```
                                    HAT_INVALID_REGION_COOKIE);
                                hat_memload_array(hat, a, pgsz, ppa, prot,
                                    hat_flag);

                                if (!(hat_flag & HAT_LOAD_LOCK)) {
                                        SEGVN_VMSTAT_FLTVNPAGES(29);
                                        for (i = 0; i < pages; i++) {
                                                page_unlock(ppa[i]);
                                        }
                                }
                                anon_array_exit(&an_cookie);
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                goto next;
                        }

                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
                            (!svd->pageprot && svd->prot == (prot & vpprot)));

                        pfn = page_pptonum(ppa[0]);
                        /*
                         * hat_page_demote() needs an SE_EXCL lock on one of
                         * constituent page_t’s and it decreases root’s p_szc
                         * last. This means if root’s p_szc is equal szc and
                         * all its constituent pages are locked
                         * hat_page_demote() that could have changed p_szc to
                         * szc is already done and no new have page_demote()
                         * can start for this large page.
                         */

                        /*
                         * we need to make sure same mapping size is used for
                         * the same address range if there’s a possibility the
                         * adddress is already mapped because hat layer panics
                         * when translation is loaded for the range already
                         * mapped with a different page size.  We achieve it
                         * by always using largest page size possible subject
                         * to the constraints of page size, segment page size
                         * and page alignment.  Since mappings are invalidated
                         * when those constraints change and make it
                         * impossible to use previously used mapping size no
                         * mapping size conflicts should happen.
                         */

                chkszc:
                        if ((pszc = ppa[0]->p_szc) == szc &&
                            IS_P2ALIGNED(pfn, pages)) {

                                SEGVN_VMSTAT_FLTVNPAGES(30);
#ifdef DEBUG
                                for (i = 0; i < pages; i++) {
                                        ASSERT(PAGE_LOCKED(ppa[i]));
                                        ASSERT(!PP_ISFREE(ppa[i]));
                                        ASSERT(page_pptonum(ppa[i]) ==
                                            pfn + i);
                                        ASSERT(ppa[i]->p_szc == szc);
                                        ASSERT(ppa[i]->p_vnode == vp);
                                        ASSERT(ppa[i]->p_offset ==
                                            off + (i << PAGESHIFT));
                                }
#endif /* DEBUG */
                                /*
                                 * All pages are of szc we need and they are
                                 * all locked so they can’t change szc. load
                                 * translations.
                                 *
                                 * if page got promoted since last check
                                 * we don’t need pplist.
                                 */
                                if (pplist != NULL) {
                                        page_free_replacement_page(pplist);
                                        page_create_putback(pages);
                                }
                                if (PP_ISMIGRATE(ppa[0])) {
                                        page_migrate(seg, a, ppa, pages);
                                }
                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
                                    prot, vpprot);
                                if (!xhat) {
                                        hat_memload_array_region(hat, a, pgsz,
                                            ppa, prot & vpprot, hat_flag,
                                            svd->rcookie);
                                } else {
                                        /*
                                         * avoid large xhat mappings to FS
                                         * pages so that hat_page_demote()
                                         * doesn’t need to check for xhat
                                         * large mappings.
                                         * Don’t use regions with xhats.
                                         */
                                        for (i = 0; i < pages; i++) {
                                                hat_memload(hat,
                                                    a + (i << PAGESHIFT),
                                                    ppa[i], prot & vpprot,
                                                    hat_flag);
                                        }
                                }

                                if (!(hat_flag & HAT_LOAD_LOCK)) {
                                        for (i = 0; i < pages; i++) {
                                                page_unlock(ppa[i]);
                                        }
                                }
                                if (amp != NULL) {
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                                goto next;
                        }

                        /*
                         * See if upsize is possible.
                         */
                        if (pszc > szc && szc < seg->s_szc &&
                            (segvn_anypgsz_vnode || pszc >= seg->s_szc)) {
                                pgcnt_t aphase;
                                uint_t pszc1 = MIN(pszc, seg->s_szc);
                                ppgsz = page_get_pagesize(pszc1);
                                ppages = btop(ppgsz);
                                aphase = btop(P2PHASE((uintptr_t)a, ppgsz));

                                ASSERT(type != F_SOFTLOCK);

                                SEGVN_VMSTAT_FLTVNPAGES(31);
                                if (aphase != P2PHASE(pfn, ppages)) {
                                        segvn_faultvnmpss_align_err4++;
                                } else {
                                        SEGVN_VMSTAT_FLTVNPAGES(32);
                                        if (pplist != NULL) {
                                                page_t *pl = pplist;
                                                page_free_replacement_page(pl);
                                                page_create_putback(pages);
                                        }
```

```
                                        for (i = 0; i < pages; i++) {
                                                page_unlock(ppa[i]);
                                        }
                                        if (amp != NULL) {
                                                anon_array_exit(&an_cookie);
                                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                        }
                                        pszc = pszc1;
                                        ierr = -2;
                                        break;
                                }
                        }

                        /*
                         * check if we should use smallest mapping size.
                         */
                        upgrdfail = 0;
                        if (szc == 0 || xhat ||
                            (pszc >= szc &&
                            !IS_P2ALIGNED(pfn, pages)) ||
                            (pszc < szc &&
                            !segvn_full_szcpages(ppa, szc, &upgrdfail,
                            &pszc))) {

                                if (upgrdfail && type != F_SOFTLOCK) {
                                        /*
                                         * segvn_full_szcpages failed to lock
                                         * all pages EXCL. Size down.
                                         */
                                        ASSERT(pszc < szc);

                                        SEGVN_VMSTAT_FLTVNPAGES(33);

                                        if (pplist != NULL) {
                                                page_t *pl = pplist;
                                                page_free_replacement_page(pl);
                                                page_create_putback(pages);
                                        }

                                        for (i = 0; i < pages; i++) {
                                                page_unlock(ppa[i]);
                                        }
                                        if (amp != NULL) {
                                                anon_array_exit(&an_cookie);
                                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                        }
                                        ierr = -1;
                                        break;
                                }
                                if (szc != 0 && !xhat && !upgrdfail) {
                                        segvn_faultvnmpss_align_err5++;
                                }
                                SEGVN_VMSTAT_FLTVNPAGES(34);
                                if (pplist != NULL) {
                                        page_free_replacement_page(pplist);
                                        page_create_putback(pages);
                                }
                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
                                    prot, vpprot);
                                if (upgrdfail && segvn_anypgsz_vnode) {
                                        /* SOFTLOCK case */
                                        hat_memload_array_region(hat, a, pgsz,
                                            ppa, prot & vpprot, hat_flag,
                                            svd->rcookie);
                                } else {
                                        for (i = 0; i < pages; i++) {
                                                hat_memload_region(hat,
                                                    a + (i << PAGESHIFT),
                                                    ppa[i], prot & vpprot,
                                                    hat_flag, svd->rcookie);
                                        }
                                }
                                if (!(hat_flag & HAT_LOAD_LOCK)) {
                                        for (i = 0; i < pages; i++) {
                                                page_unlock(ppa[i]);
                                        }
                                }
                                if (amp != NULL) {
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                                goto next;
                        }

                        if (pszc == szc) {
                                /*
                                 * segvn_full_szcpages() upgraded pages szc.
                                 */
                                ASSERT(pszc == ppa[0]->p_szc);
                                ASSERT(IS_P2ALIGNED(pfn, pages));
                                goto chkszc;
                        }

                        if (pszc > szc) {
                                kmutex_t *szcmtx;
                                SEGVN_VMSTAT_FLTVNPAGES(35);
                                /*
                                 * p_szc of ppa[0] can change since we haven't
                                 * locked all constituent pages. Call
                                 * page_lock_szc() to prevent szc changes.
                                 * This should be a rare case that happens when
                                 * multiple segments use a different page size
                                 * to map the same file offsets.
                                 */
                                szcmtx = page_szc_lock(ppa[0]);
                                pszc = ppa[0]->p_szc;
                                ASSERT(szcmtx != NULL || pszc == 0);
                                ASSERT(ppa[0]->p_szc <= pszc);
                                if (pszc <= szc) {
                                        SEGVN_VMSTAT_FLTVNPAGES(36);
                                        if (szcmtx != NULL) {
                                                mutex_exit(szcmtx);
                                        }
                                        goto chkszc;
                                }
                                if (pplist != NULL) {
                                        /*
                                         * page got promoted since last check.
                                         * we don't need preaalocated large
                                         * page.
                                         */
                                        SEGVN_VMSTAT_FLTVNPAGES(37);
                                        page_free_replacement_page(pplist);
                                        page_create_putback(pages);
                                }
                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
                                    prot, vpprot);
                                hat_memload_array_region(hat, a, pgsz, ppa,
                                    prot & vpprot, hat_flag, svd->rcookie);
                                mutex_exit(szcmtx);
                                if (!(hat_flag & HAT_LOAD_LOCK)) {
                                        for (i = 0; i < pages; i++) {
```

```
                                                page_unlock(ppa[i]);
                                        }
                                }
                                if (amp != NULL) {
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                                goto next;
                        }

                        /*
                         * if page got demoted since last check
                         * we could have not allocated larger page.
                         * allocate now.
                         */
                        if (pplist == NULL &&
                            page_alloc_pages(vp, seg, a, &pplist, NULL,
                            szc, 0, 0) && type != F_SOFTLOCK) {
                                SEGVN_VMSTAT_FLTVNPAGES(38);
                                for (i = 0; i < pages; i++) {
                                        page_unlock(ppa[i]);
                                }
                                if (amp != NULL) {
                                        anon_array_exit(&an_cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                                ierr = -1;
                                alloc_failed |= (1 << szc);
                                break;
                        }

                        SEGVN_VMSTAT_FLTVNPAGES(39);

                        if (pplist != NULL) {
                                segvn_relocate_pages(ppa, pplist);
#ifdef DEBUG
                        } else {
                                ASSERT(type == F_SOFTLOCK);
                                SEGVN_VMSTAT_FLTVNPAGES(40);
#endif /* DEBUG */
                        }

                        SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot);

                        if (pplist == NULL && segvn_anypgsz_vnode == 0) {
                                ASSERT(type == F_SOFTLOCK);
                                for (i = 0; i < pages; i++) {
                                        ASSERT(ppa[i]->p_szc < szc);
                                        hat_memload_region(hat,
                                            a + (i << PAGESHIFT),
                                            ppa[i], prot & vpprot, hat_flag,
                                            svd->rcookie);
                                }
                        } else {
                                ASSERT(pplist != NULL || type == F_SOFTLOCK);
                                hat_memload_array_region(hat, a, pgsz, ppa,
                                    prot & vpprot, hat_flag, svd->rcookie);
                        }
                        if (!(hat_flag & HAT_LOAD_LOCK)) {
                                for (i = 0; i < pages; i++) {
                                        ASSERT(PAGE_SHARED(ppa[i]));
                                        page_unlock(ppa[i]);
                                }
                        }
                        if (amp != NULL) {
                                anon_array_exit(&an_cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }

        next:
                if (vpage != NULL) {
                        vpage += pages;
                }
                adjszc_chk = 1;
        }
        if (a == lpgeaddr)
                break;
        ASSERT(a < lpgeaddr);

        ASSERT(!brkcow && !tron && type != F_SOFTLOCK);

        /*
         * ierr == -1 means we failed to map with a large page.
         * (either due to allocation/relocation failures or
         * misalignment with other mappings to this file.
         *
         * ierr == -2 means some other thread allocated a large page
         * after we gave up tp map with a large page.  retry with
         * larger mapping.
         */
        ASSERT(ierr == -1 || ierr == -2);
        ASSERT(ierr == -2 || szc != 0);
        ASSERT(ierr == -1 || szc < seg->s_szc);
        if (ierr == -2) {
                SEGVN_VMSTAT_FLTVNPAGES(41);
                ASSERT(pszc > szc && pszc <= seg->s_szc);
                szc = pszc;
        } else if (segvn_anypgsz_vnode) {
                SEGVN_VMSTAT_FLTVNPAGES(42);
                szc--;
        } else {
                SEGVN_VMSTAT_FLTVNPAGES(43);
                ASSERT(pszc < szc);
                /*
                 * other process created pszc large page.
                 * but we still have to drop to 0 szc.
                 */
                szc = 0;
        }

        pgsz = page_get_pagesize(szc);
        pages = btop(pgsz);
        if (ierr == -2) {
                /*
                 * Size up case. Note lpgaddr may only be needed for
                 * softlock case so we don’t adjust it here.
                 */
                a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
                ASSERT(a >= lpgaddr);
                lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
                off = svd->offset + (uintptr_t)(a - seg->s_base);
                aindx = svd->anon_index + seg_page(seg, a);
                vpage = (svd->vpage != NULL) ?
                    &svd->vpage[seg_page(seg, a)] : NULL;
        } else {
                /*
                 * Size down case. Note lpgaddr may only be needed for
                 * softlock case so we don’t adjust it here.
                 */
                ASSERT(IS_P2ALIGNED(a, pgsz));
                ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
                lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
```

```
                        ASSERT(a < lpgeaddr);
                        if (a < addr) {
                                SEGVN_VMSTAT_FLTVNPAGES(44);
                                /*
                                 * The beginning of the large page region can
                                 * be pulled to the right to make a smaller
                                 * region. We haven’t yet faulted a single
                                 * page.
                                 */
                                a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
                                ASSERT(a >= lpgaddr);
                                off = svd->offset +
                                    (uintptr_t)(a - seg->s_base);
                                aindx = svd->anon_index + seg_page(seg, a);
                                vpage = (svd->vpage != NULL) ?
                                    &svd->vpage[seg_page(seg, a)] : NULL;
                        }
                }
        }
out:
        kmem_free(ppa, ppasize);
        if (!err && !vop_size_err) {
                SEGVN_VMSTAT_FLTVNPAGES(45);
                return (0);
        }
        if (type == F_SOFTLOCK && a > lpgaddr) {
                SEGVN_VMSTAT_FLTVNPAGES(46);
                segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
        }
        if (!vop_size_err) {
                SEGVN_VMSTAT_FLTVNPAGES(47);
                return (err);
        }
        ASSERT(brkcow || tron || type == F_SOFTLOCK);
        /*
         * Large page end is mapped beyond the end of file and it’s a cow
         * fault (can be a text replication induced cow) or softlock so we can’t
         * reduce the map area.  For now just demote the segment. This should
         * really only happen if the end of the file changed after the mapping
         * was established since when large page segments are created we make
         * sure they don’t extend beyond the end of the file.
         */
        SEGVN_VMSTAT_FLTVNPAGES(48);

        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
        err = 0;
        if (seg->s_szc != 0) {
                segvn_fltvnpages_clrszc_cnt++;
                ASSERT(svd->softlockcnt == 0);
                err = segvn_clrszc(seg);
                if (err != 0) {
                        segvn_fltvnpages_clrszc_err++;
                }
        }
        ASSERT(err || seg->s_szc == 0);
        SEGVN_LOCK_DOWNGRADE(seg->s_as, &svd->lock);
        /* segvn_fault will do its job as if szc had been zero to begin with */
        return (err == 0 ? IE_RETRY : FC_MAKE_ERR(err));
}

/*
 * This routine will attempt to fault in one large page.
 * it will use smaller pages if that fails.
 * It should only be called for pure anonymous segments.
 */
```

```
static faultcode_t
segvn_fault_anonpages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
    caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
    caddr_t eaddr, int brkcow)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp = svd->amp;
        uchar_t segtype = svd->type;
        uint_t szc = seg->s_szc;
        size_t pgsz = page_get_pagesize(szc);
        size_t maxpgsz = pgsz;
        pgcnt_t pages = btop(pgsz);
        uint_t ppaszc = szc;
        caddr_t a = lpgaddr;
        ulong_t aindx = svd->anon_index + seg_page(seg, a);
        struct vpage *vpage = (svd->vpage != NULL) ?
            &svd->vpage[seg_page(seg, a)] : NULL;
        page_t **ppa;
        uint_t  ppa_szc;
        faultcode_t err;
        int ierr;
        uint_t protchk, prot, vpprot;
        ulong_t i;
        int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
        anon_sync_obj_t cookie;
        int adjszc_chk;
        int pgflags = (svd->tr_state == SEGVN_TR_ON) ? PG_LOCAL : 0;

        ASSERT(szc != 0);
        ASSERT(amp != NULL);
        ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
        ASSERT(!(svd->flags & MAP_NORESERVE));
        ASSERT(type != F_SOFTUNLOCK);
        ASSERT(IS_P2ALIGNED(a, maxpgsz));
        ASSERT(!brkcow || svd->tr_state == SEGVN_TR_OFF);
        ASSERT(svd->tr_state != SEGVN_TR_INIT);

        ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

        VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnvmstats.fltanpages[0]);
        VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnvmstats.fltanpages[1]);

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

        if (svd->pageprot) {
                switch (rw) {
                case S_READ:
                        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;
                }
                VM_STAT_ADD(segvnvmstats.fltanpages[2]);
        } else {
                prot = svd->prot;
                /* caller has already done segment level protection check. */
```

```
        }

        ppa = kmem_cache_alloc(segvn_szc_cache[ppaszc], KM_SLEEP);
        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
        for (;;) {
                adjszc_chk = 0;
                for (; a < lpgeaddr; a += pgsz, aindx += pages) {
                        if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
                                VM_STAT_ADD(segvnvmstats.fltanpages[3]);
                                ASSERT(vpage != NULL);
                                prot = VPP_PROT(vpage);
                                ASSERT(sameprot(seg, a, maxpgsz));
                                if ((prot & protchk) == 0) {
                                        err = FC_PROT;
                                        goto error;
                                }
                        }
                        if (adjszc_chk && IS_P2ALIGNED(a, maxpgsz) &&
                            pgsz < maxpgsz) {
                                ASSERT(a > lpgaddr);
                                szc = seg->s_szc;
                                pgsz = maxpgsz;
                                pages = btop(pgsz);
                                ASSERT(IS_P2ALIGNED(aindx, pages));
                                lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr,
                                    pgsz);
                        }
                        if (type == F_SOFTLOCK) {
                                atomic_add_long((ulong_t *)&svd->softlockcnt,
                                    pages);
                        }
                        anon_array_enter(amp, aindx, &cookie);
                        ppa_szc = (uint_t)-1;
                        ierr = anon_map_getpages(amp, aindx, szc, seg, a,
                            prot, &vpprot, ppa, &ppa_szc, vpage, rw, brkcow,
                            segvn_anypgsz, pgflags, svd->cred);
                        if (ierr != 0) {
                                anon_array_exit(&cookie);
                                VM_STAT_ADD(segvnvmstats.fltanpages[4]);
                                if (type == F_SOFTLOCK) {
                                        atomic_add_long(
                                            (ulong_t *)&svd->softlockcnt,
                                            -pages);
                                }
                                if (ierr > 0) {
                                        VM_STAT_ADD(segvnvmstats.fltanpages[6]);
                                        err = FC_MAKE_ERR(ierr);
                                        goto error;
                                }
                                break;
                        }

                        ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));

                        ASSERT(segtype == MAP_SHARED ||
                            ppa[0]->p_szc <= szc);
                        ASSERT(segtype == MAP_PRIVATE ||
                            ppa[0]->p_szc >= szc);

                        /*
                         * Handle pages that have been marked for migration
                         */
                        if (lgrp_optimizations())
                                page_migrate(seg, a, ppa, pages);

                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                        if (segtype == MAP_SHARED) {
                                vpprot |= PROT_WRITE;
                        }

                        hat_memload_array(hat, a, pgsz, ppa,
                            prot & vpprot, hat_flag);

                        if (hat_flag & HAT_LOAD_LOCK) {
                                VM_STAT_ADD(segvnvmstats.fltanpages[7]);
                        } else {
                                VM_STAT_ADD(segvnvmstats.fltanpages[8]);
                                for (i = 0; i < pages; i++)
                                        page_unlock(ppa[i]);
                        }
                        if (vpage != NULL)
                                vpage += pages;

                        anon_array_exit(&cookie);
                        adjszc_chk = 1;
                }
                if (a == lpgeaddr)
                        break;
                ASSERT(a < lpgeaddr);
                /*
                 * ierr == -1 means we failed to allocate a large page.
                 * so do a size down operation.
                 *
                 * ierr == -2 means some other process that privately shares
                 * pages with this process has allocated a larger page and we
                 * need to retry with larger pages. So do a size up
                 * operation. This relies on the fact that large pages are
                 * never partially shared i.e. if we share any constituent
                 * page of a large page with another process we must share the
                 * entire large page. Note this cannot happen for SOFTLOCK
                 * case, unless current address (a) is at the beginning of the
                 * next page size boundary because the other process couldn't
                 * have relocated locked pages.
                 */
                ASSERT(ierr == -1 || ierr == -2);

                if (segvn_anypgsz) {
                        ASSERT(ierr == -2 || szc != 0);
                        ASSERT(ierr == -1 || szc < seg->s_szc);
                        szc = (ierr == -1) ? szc - 1 : szc + 1;
                } else {
                        /*
                         * For non COW faults and segvn_anypgsz == 0
                         * we need to be careful not to loop forever
                         * if existing page is found with szc other
                         * than 0 or seg->s_szc. This could be due
                         * to page relocations on behalf of DR or
                         * more likely large page creation. For this
                         * case simply re-size to existing page's szc
                         * if returned by anon_map_getpages().
                         */
                        if (ppa_szc == (uint_t)-1) {
                                szc = (ierr == -1) ? 0 : seg->s_szc;
                        } else {
                                ASSERT(ppa_szc <= seg->s_szc);
                                ASSERT(ierr == -2 || ppa_szc < szc);
                                ASSERT(ierr == -1 || ppa_szc > szc);
                                szc = ppa_szc;
                        }
                }
```

```
                pgsz = page_get_pagesize(szc);
                pages = btop(pgsz);
                ASSERT(type != F_SOFTLOCK || ierr == -1 ||
                    (IS_P2ALIGNED(a, pgsz) && IS_P2ALIGNED(lpgeaddr, pgsz)));
                if (type == F_SOFTLOCK) {
                        /*
                         * For softlocks we cannot reduce the fault area
                         * (calculated based on the largest page size for this
                         * segment) for size down and a is already next
                         * page size aligned as assertted above for size
                         * ups. Therefore just continue in case of softlock.
                         */
                        VM_STAT_ADD(segvnvmstats.fltanpages[9]);
                        continue; /* keep lint happy */
                } else if (ierr == -2) {

                        /*
                         * Size up case. Note lpgaddr may only be needed for
                         * softlock case so we don't adjust it here.
                         */
                        VM_STAT_ADD(segvnvmstats.fltanpages[10]);
                        a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
                        ASSERT(a >= lpgaddr);
                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
                        aindx = svd->anon_index + seg_page(seg, a);
                        vpage = (svd->vpage != NULL) ?
                            &svd->vpage[seg_page(seg, a)] : NULL;
                } else {
                        /*
                         * Size down case. Note lpgaddr may only be needed for
                         * softlock case so we don't adjust it here.
                         */
                        VM_STAT_ADD(segvnvmstats.fltanpages[11]);
                        ASSERT(IS_P2ALIGNED(a, pgsz));
                        ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
                        ASSERT(a < lpgeaddr);
                        if (a < addr) {
                                /*
                                 * The beginning of the large page region can
                                 * be pulled to the right to make a smaller
                                 * region. We haven't yet faulted a single
                                 * page.
                                 */
                                VM_STAT_ADD(segvnvmstats.fltanpages[12]);
                                a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
                                ASSERT(a >= lpgaddr);
                                aindx = svd->anon_index + seg_page(seg, a);
                                vpage = (svd->vpage != NULL) ?
                                    &svd->vpage[seg_page(seg, a)] : NULL;
                        }
                }
        }
        VM_STAT_ADD(segvnvmstats.fltanpages[13]);
        ANON_LOCK_EXIT(&amp->a_rwlock);
        kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
        return (0);
error:
        VM_STAT_ADD(segvnvmstats.fltanpages[14]);
        ANON_LOCK_EXIT(&amp->a_rwlock);
        kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
        if (type == F_SOFTLOCK && a > lpgaddr) {
                VM_STAT_ADD(segvnvmstats.fltanpages[15]);
                segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
        }
        return (err);
```

```
}

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[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++;
                                        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);
                                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 > 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)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);
}

/*
 * This routine is used to start I/O on pages asynchronously.  XXX it will
 * only create PAGESIZE pages. At fault time they will be relocated into
 * larger pages.
 */
static faultcode_t
segvn_faulta(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        int err;
        struct anon_map *amp;
        vnode_t *vp;
```

```
        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        if ((amp = svd->amp) != NULL) {
                struct anon *ap;

                /*
                 * Reader lock to prevent amp->ahp from being changed.
                 * This is advisory, it's ok to miss a page, so
                 * we don't do anon_array_enter lock.
                 */
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                if ((ap = anon_get_ptr(amp->ahp,
                    svd->anon_index + seg_page(seg, addr))) != NULL) {

                        err = anon_getpage(&ap, NULL, NULL,
                            0, seg, addr, S_READ, svd->cred);

                        ANON_LOCK_EXIT(&amp->a_rwlock);
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        if (err)
                                return (FC_MAKE_ERR(err));
                        return (0);
                }
                ANON_LOCK_EXIT(&amp->a_rwlock);
        }

        if (svd->vp == NULL) {
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);                     /* zfod page - do nothing now */
        }

        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)(svd->offset + (uintptr_t)(addr - seg->s_base)),
            PAGESIZE, NULL, NULL, 0, seg, addr,
            S_OTHER, svd->cred, NULL);

        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        if (err)
                return (FC_MAKE_ERR(err));
        return (0);
}

static int
segvn_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *cvp, *svp, *evp;
        struct vnode *vp;
        size_t pgsz;
        pgcnt_t pgcnt;
        anon_sync_obj_t cookie;
        int unload_done = 0;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        if ((svd->maxprot & prot) != prot)
                return (EACCES);                        /* violated maxprot */

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

        /* return if prot is the same */
        if (!svd->pageprot && svd->prot == prot) {
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);
        }

        /*
         * Since we change protections we first have to flush the cache.
         * This makes sure all the pagelock calls have to recheck
         * protections.
         */
        if (svd->softlockcnt > 0) {
                ASSERT(svd->tr_state == SEGVN_TR_OFF);

                /*
                 * If this is shared segment non 0 softlockcnt
                 * means locked pages are still in use.
                 */
                if (svd->type == MAP_SHARED) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EAGAIN);
                }

                /*
                 * Since we do have the segvn writers lock nobody can fill
                 * the cache with entries belonging to this seg during
                 * the purge. The flush either succeeds or we still have
                 * pending I/Os.
                 */
                segvn_purge(seg);
                if (svd->softlockcnt > 0) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EAGAIN);
                }
        }

        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(svd->amp == NULL);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                    HAT_REGION_TEXT);
                svd->rcookie = HAT_INVALID_REGION_COOKIE;
                unload_done = 1;
        } else if (svd->tr_state == SEGVN_TR_INIT) {
                svd->tr_state = SEGVN_TR_OFF;
        } else if (svd->tr_state == SEGVN_TR_ON) {
                ASSERT(svd->amp != NULL);
                segvn_textunrepl(seg, 0);
                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
                unload_done = 1;
        }

        if ((prot & PROT_WRITE) && svd->type == MAP_SHARED &&
            svd->vp != NULL && (svd->vp->v_flag & VVMEXEC)) {
                ASSERT(vn_is_mapped(svd->vp, V_WRITE));
                segvn_inval_trcache(svd->vp);
        }
        if (seg->s_szc != 0) {
                int err;
                pgsz = page_get_pagesize(seg->s_szc);
                pgcnt = pgsz >> PAGESHIFT;
                ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
                if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        ASSERT(seg->s_base != addr || seg->s_size != len);
                        /*
                         * If we are holding the as lock as a reader then
                         * we need to return IE_RETRY and let the as
```

```
                         * layer drop and re-acquire the lock as a writer.
                         */
                        if (AS_READ_HELD(seg->s_as, &seg->s_as->a_lock))
                                return (IE_RETRY);
                        VM_STAT_ADD(segvnvmstats.demoterange[1]);
                        if (svd->type == MAP_PRIVATE || svd->vp != NULL) {
                                err = segvn_demote_range(seg, addr, len,
                                    SDR_END, 0);
                        } else {
                                uint_t szcvec = map_pgszcvec(seg->s_base,
                                    pgsz, (uintptr_t)seg->s_base,
                                    (svd->flags & MAP_TEXT), MAPPGSZC_SHM, 0);
                                err = segvn_demote_range(seg, addr, len,
                                    SDR_END, szcvec);
                        }
                        if (err == 0)
                                return (IE_RETRY);
                        if (err == ENOMEM)
                                return (IE_NOMEM);
                        return (err);
                }
        }


        /*
         * If it's a private mapping and we're making it writable then we
         * may have to reserve the additional swap space now. If we are
         * making writable only a part of the segment then we use its vpage
         * array to keep a record of the pages for which we have reserved
         * swap. In this case we set the pageswap field in the segment's
         * segvn structure to record this.
         *
         * If it's a private mapping to a file (i.e., vp != NULL) and we're
         * removing write permission on the entire segment and we haven't
         * modified any pages, we can release the swap space.
         */
        if (svd->type == MAP_PRIVATE) {
                if (prot & PROT_WRITE) {
                        if (!(svd->flags & MAP_NORESERVE) &&
                            !(svd->swresv && svd->pageswap == 0)) {
                                size_t sz = 0;

                                /*
                                 * Start by determining how much swap
                                 * space is required.
                                 */
                                if (addr == seg->s_base &&
                                    len == seg->s_size &&
                                    svd->pageswap == 0) {
                                        /* The whole segment */
                                        sz = seg->s_size;
                                } else {
                                        /*
                                         * Make sure that the vpage array
                                         * exists, and make a note of the
                                         * range of elements corresponding
                                         * to len.
                                         */
                                        segvn_vpage(seg);
                                        svp = &svd->vpage[seg_page(seg, addr)];
                                        evp = &svd->vpage[seg_page(seg,
                                            addr + len)];

                                        if (svd->pageswap == 0) {
                                                /*
                                                 * This is the first time we've
                                                 * asked for a part of this
                                                 * segment, so we need to
                                                 * reserve everything we’ve
                                                 * been asked for.
                                                 */
                                                sz = len;
                                        } else {
                                                /*
                                                 * We have to count the number
                                                 * of pages required.
                                                 */
                                                for (cvp = svp;  cvp < evp;
                                                    cvp++) {
                                                        if (!VPP_ISSWAPRES(cvp))
                                                                sz++;
                                                }
                                                sz <<= PAGESHIFT;
                                        }
                                }

                                /* Try to reserve the necessary swap. */
                                if (anon_resv_zone(sz,
                                    seg->s_as->a_proc->p_zone) == 0) {
                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                        return (IE_NOMEM);
                                }

                                /*
                                 * Make a note of how much swap space
                                 * we’ve reserved.
                                 */
                                if (svd->pageswap == 0 && sz == seg->s_size) {
                                        svd->swresv = sz;
                                } else {
                                        ASSERT(svd->vpage != NULL);
                                        svd->swresv += sz;
                                        svd->pageswap = 1;
                                        for (cvp = svp; cvp < evp; cvp++) {
                                                if (!VPP_ISSWAPRES(cvp))
                                                        VPP_SETSWAPRES(cvp);
                                        }
                                }
                        }
                } else {
                        /*
                         * Swap space is released only if this segment
                         * does not map anonymous memory, since read faults
                         * on such segments still need an anon slot to read
                         * in the data.
                         */
                        if (svd->swresv != 0 && svd->vp != NULL &&
                            svd->amp == NULL && addr == seg->s_base &&
                            len == seg->s_size && svd->pageprot == 0) {
                                ASSERT(svd->pageswap == 0);
                                anon_unresv_zone(svd->swresv,
                                    seg->s_as->a_proc->p_zone);
                                svd->swresv = 0;
                                TRACE_3(TR_FAC_VM, TR_ANON_PROC,
                                    "anon proc:%p %lu %u", seg, 0, 0);
                        }
                }
        }

        if (addr == seg->s_base && len == seg->s_size && svd->vpage == NULL) {
                if (svd->prot == prot) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
```

```
                        return (0);                     /* all done */
                }
                svd->prot = (uchar_t)prot;
        } else if (svd->type == MAP_PRIVATE) {
                struct anon *ap = NULL;
                page_t *pp;
                u_offset_t offset, off;
                struct anon_map *amp;
                ulong_t anon_idx = 0;

                /*
                 * A vpage structure exists or else the change does not
                 * involve the entire segment.  Establish a vpage structure
                 * if none is there.  Then, for each page in the range,
                 * adjust its individual permissions.  Note that write-
                 * enabling a MAP_PRIVATE page can affect the claims for
                 * locked down memory.  Overcommitting memory terminates
                 * the operation.
                 */
                segvn_vpage(seg);
                svd->pageprot = 1;
                if ((amp = svd->amp) != NULL) {
                        anon_idx = svd->anon_index + seg_page(seg, addr);
                        ASSERT(seg->s_szc == 0 ||
                            IS_P2ALIGNED(anon_idx, pgcnt));
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                }

                offset = svd->offset + (uintptr_t)(addr - seg->s_base);
                evp = &svd->vpage[seg_page(seg, addr + len)];

                /*
                 * See Statement at the beginning of segvn_lockop regarding
                 * the way cowcnts and lckcnts are handled.
                 */
                for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {

                        if (seg->s_szc != 0) {
                                if (amp != NULL) {
                                        anon_array_enter(amp, anon_idx,
                                            &cookie);
                                }
                                if (IS_P2ALIGNED(anon_idx, pgcnt) &&
                                    !segvn_claim_pages(seg, svp, offset,
                                    anon_idx, prot)) {
                                        if (amp != NULL) {
                                                anon_array_exit(&cookie);
                                        }
                                        break;
                                }
                                if (amp != NULL) {
                                        anon_array_exit(&cookie);
                                }
                                anon_idx++;
                        } else {
                                if (amp != NULL) {
                                        anon_array_enter(amp, anon_idx,
                                            &cookie);
                                        ap = anon_get_ptr(amp->ahp, anon_idx++);
                                }

                                if (VPP_ISPPLOCK(svp) &&
                                    VPP_PROT(svp) != prot) {

                                        if (amp == NULL || ap == NULL) {
                                                vp = svd->vp;
                                                off = offset;
                                        } else
                                                swap_xlate(ap, &vp, &off);
                                        if (amp != NULL)
                                                anon_array_exit(&cookie);

                                        if ((pp = page_lookup(vp, off,
                                            SE_SHARED)) == NULL) {
                                                panic("segvn_setprot: no page");
                                                /*NOTREACHED*/
                                        }
                                        ASSERT(seg->s_szc == 0);
                                        if ((VPP_PROT(svp) ^ prot) &
                                            PROT_WRITE) {
                                                if (prot & PROT_WRITE) {
                                                        if (!page_addclaim(
                                                            pp)) {
                                                                page_unlock(pp);
                                                                break;
                                                        }
                                                } else {
                                                        if (!page_subclaim(
                                                            pp)) {
                                                                page_unlock(pp);
                                                                break;
                                                        }
                                                }
                                        }
                                        page_unlock(pp);
                                } else if (amp != NULL)
                                        anon_array_exit(&cookie);
                        }
                        VPP_SETPROT(svp, prot);
                        offset += PAGESIZE;
                }
                if (amp != NULL)
                        ANON_LOCK_EXIT(&amp->a_rwlock);

                /*
                 * Did we terminate prematurely?  If so, simply unload
                 * the translations to the things we've updated so far.
                 */
                if (svp != evp) {
                        if (unload_done) {
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                return (IE_NOMEM);
                        }
                        len = (svp - &svd->vpage[seg_page(seg, addr)]) *
                            PAGESIZE;
                        ASSERT(seg->s_szc == 0 || IS_P2ALIGNED(len, pgsz));
                        if (len != 0)
                                hat_unload(seg->s_as->a_hat, addr,
                                    len, HAT_UNLOAD);
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (IE_NOMEM);
                }
        } else {
                segvn_vpage(seg);
                svd->pageprot = 1;
                evp = &svd->vpage[seg_page(seg, addr + len)];
                for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {
                        VPP_SETPROT(svp, prot);
                }
        }

        if (unload_done) {
```

```
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);
        }

        if (((prot & PROT_WRITE) != 0 &&
            (svd->vp != NULL || svd->type == MAP_PRIVATE)) ||
            (prot & ~PROT_USER) == PROT_NONE) {
                /*
                 * Either private or shared data with write access (in
                 * which case we need to throw out all former translations
                 * so that we get the right translations set up on fault
                 * and we don't allow write access to any copy-on-write pages
                 * that might be around or to prevent write access to pages
                 * representing holes in a file), or we don't have permission
                 * to access the memory at all (in which case we have to
                 * unload any current translations that might exist).
                 */
                hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
        } else {
                /*
                 * A shared mapping or a private mapping in which write
                 * protection is going to be denied - just change all the
                 * protections over the range of addresses in question.
                 * segvn does not support any other attributes other
                 * than prot so we can use hat_chgattr.
                 */
                hat_chgattr(seg->s_as->a_hat, addr, len, prot);
        }

        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

        return (0);
}

/*
 * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
 * to determine if the seg is capable of mapping the requested szc.
 */
static int
segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct segvn_data *nsvd;
        struct anon_map *amp = svd->amp;
        struct seg *nseg;
        caddr_t eaddr = addr + len, a;
        size_t pgsz = page_get_pagesize(szc);
        pgcnt_t pgcnt = page_get_pagecnt(szc);
        int err;
        u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);

        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);

        if (seg->s_szc == szc || segvn_lpg_disable != 0) {
                return (0);
        }

        /*
         * addr should always be pgsz aligned but eaddr may be misaligned if
         * it's at the end of the segment.
         *
         * XXX we should assert this condition since as_setpagesize() logic
         * guarantees it.
         */
        if (!IS_P2ALIGNED(addr, pgsz) ||
            (!IS_P2ALIGNED(eaddr, pgsz) &&
            eaddr != seg->s_base + seg->s_size)) {

                segvn_setpgsz_align_err++;
                return (EINVAL);
        }

        if (amp != NULL && svd->type == MAP_SHARED) {
                ulong_t an_idx = svd->anon_index + seg_page(seg, addr);
                if (!IS_P2ALIGNED(an_idx, pgcnt)) {

                        segvn_setpgsz_anon_align_err++;
                        return (EINVAL);
                }
        }

        if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
            szc > segvn_maxpgszc) {
                return (EINVAL);
        }

        /* paranoid check */
        if (svd->vp != NULL &&
            (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
                return (EINVAL);
        }

        if (seg->s_szc == 0 && svd->vp != NULL &&
            map_addr_vacalign_check(addr, off)) {
                return (EINVAL);
        }

        /*
         * Check that protections are the same within new page
         * size boundaries.
         */
        if (svd->pageprot) {
                for (a = addr; a < eaddr; a += pgsz) {
                        if ((a + pgsz) > eaddr) {
                                if (!sameprot(seg, a, eaddr - a)) {
                                        return (EINVAL);
                                }
                        } else {
                                if (!sameprot(seg, a, pgsz)) {
                                        return (EINVAL);
                                }
                        }
                }
        }

        /*
         * Since we are changing page size we first have to flush
         * the cache. This makes sure all the pagelock calls have
         * to recheck protections.
         */
        if (svd->softlockcnt > 0) {
                ASSERT(svd->tr_state == SEGVN_TR_OFF);

                /*
                 * If this is shared segment non 0 softlockcnt
                 * means locked pages are still in use.
                 */
                if (svd->type == MAP_SHARED) {
                        return (EAGAIN);
                }
```

```
                /*
                 * Since we do have the segvn writers lock nobody can fill
                 * the cache with entries belonging to this seg during
                 * the purge. The flush either succeeds or we still have
                 * pending I/Os.
                 */
                segvn_purge(seg);
                if (svd->softlockcnt > 0) {
                        return (EAGAIN);
                }
        }

        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(svd->amp == NULL);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                    HAT_REGION_TEXT);
                svd->rcookie = HAT_INVALID_REGION_COOKIE;
        } else if (svd->tr_state == SEGVN_TR_INIT) {
                svd->tr_state = SEGVN_TR_OFF;
        } else if (svd->tr_state == SEGVN_TR_ON) {
                ASSERT(svd->amp != NULL);
                segvn_textunrepl(seg, 1);
                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
                amp = NULL;
        }

        /*
         * Operation for sub range of existing segment.
         */
        if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
                if (szc < seg->s_szc) {
                        VM_STAT_ADD(segvnvmstats.demoterange[2]);
                        err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
                        if (err == 0) {
                                return (IE_RETRY);
                        }
                        if (err == ENOMEM) {
                                return (IE_NOMEM);
                        }
                        return (err);
                }
                if (addr != seg->s_base) {
                        nseg = segvn_split_seg(seg, addr);
                        if (eaddr != (nseg->s_base + nseg->s_size)) {
                                /* eaddr is szc aligned */
                                (void) segvn_split_seg(nseg, eaddr);
                        }
                        return (IE_RETRY);
                }
                if (eaddr != (seg->s_base + seg->s_size)) {
                        /* eaddr is szc aligned */
                        (void) segvn_split_seg(seg, eaddr);
                }
                return (IE_RETRY);
        }

        /*
         * Break any low level sharing and reset seg->s_szc to 0.
         */
        if ((err = segvn_clrszc(seg)) != 0) {
                if (err == ENOMEM) {
                        err = IE_NOMEM;
                }
                return (err);
        }
        ASSERT(seg->s_szc == 0);

        /*
         * If the end of the current segment is not pgsz aligned
         * then attempt to concatenate with the next segment.
         */
        if (!IS_P2ALIGNED(eaddr, pgsz)) {
                nseg = AS_SEGNEXT(seg->s_as, seg);
                if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
                        return (ENOMEM);
                }
                if (nseg->s_ops != &segvn_ops) {
                        return (EINVAL);
                }
                nsvd = (struct segvn_data *)nseg->s_data;
                if (nsvd->softlockcnt > 0) {
                        /*
                         * If this is shared segment non 0 softlockcnt
                         * means locked pages are still in use.
                         */
                        if (nsvd->type == MAP_SHARED) {
                                return (EAGAIN);
                        }
                        segvn_purge(nseg);
                        if (nsvd->softlockcnt > 0) {
                                return (EAGAIN);
                        }
                }
                err = segvn_clrszc(nseg);
                if (err == ENOMEM) {
                        err = IE_NOMEM;
                }
                if (err != 0) {
                        return (err);
                }
                ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
                err = segvn_concat(seg, nseg, 1);
                if (err == -1) {
                        return (EINVAL);
                }
                if (err == -2) {
                        return (IE_NOMEM);
                }
                return (IE_RETRY);
        }

        /*
         * May need to re-align anon array to
         * new szc.
         */
        if (amp != NULL) {
                if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
                        struct anon_hdr *nahp;

                        ASSERT(svd->type == MAP_PRIVATE);

                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                        ASSERT(amp->refcnt == 1);
                        nahp = anon_create(btop(amp->size), ANON_NOSLEEP);
                        if (nahp == NULL) {
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                return (IE_NOMEM);
                        }
                        if (anon_copy_ptr(amp->ahp, svd->anon_index,
                            nahp, 0, btop(seg->s_size), ANON_NOSLEEP)) {
                                anon_release(nahp, btop(amp->size));
```

```
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                return (IE_NOMEM);
                        }
                        anon_release(amp->ahp, btop(amp->size));
                        amp->ahp = nahp;
                        svd->anon_index = 0;
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }
        }
        if (svd->vp != NULL && szc != 0) {
                struct vattr va;
                u_offset_t eoffpage = svd->offset;
                va.va_mask = AT_SIZE;
                eoffpage += seg->s_size;
                eoffpage = btopr(eoffpage);
                if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
                        segvn_setpgsz_getattr_err++;
                        return (EINVAL);
                }
                if (btopr(va.va_size) < eoffpage) {
                        segvn_setpgsz_eof_err++;
                        return (EINVAL);
                }
                if (amp != NULL) {
                        /*
                         * anon_fill_cow_holes() may call VOP_GETPAGE().
                         * don’t take anon map lock here to avoid holding it
                         * across VOP_GETPAGE() calls that may call back into
                         * segvn for klsutering checks. We don’t really need
                         * anon map lock here since it’s a private segment and
                         * we hold as level lock as writers.
                         */
                        if ((err = anon_fill_cow_holes(seg, seg->s_base,
                            amp->ahp, svd->anon_index, svd->vp, svd->offset,
                            seg->s_size, szc, svd->prot, svd->vpage,
                            svd->cred)) != 0) {
                                return (EINVAL);
                        }
                }
                segvn_setvnode_mpss(svd->vp);
        }

        if (amp != NULL) {
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
                if (svd->type == MAP_PRIVATE) {
                        amp->a_szc = szc;
                } else if (szc > amp->a_szc) {
                        amp->a_szc = szc;
                }
                ANON_LOCK_EXIT(&amp->a_rwlock);
        }

        seg->s_szc = szc;

        return (0);
}

static int
segvn_clrszc(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp = svd->amp;
        size_t pgsz;
        pgcnt_t pages;
        int err = 0;
        caddr_t a = seg->s_base;
        caddr_t ea = a + seg->s_size;
        ulong_t an_idx = svd->anon_index;
        vnode_t *vp = svd->vp;
        struct vpage *vpage = svd->vpage;
        page_t *anon_pl[1 + 1], *pp;
        struct anon *ap, *oldap;
        uint_t prot = svd->prot, vpprot;
        int pageflag = 0;

        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
            SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
        ASSERT(svd->softlockcnt == 0);

        if (vp == NULL && amp == NULL) {
                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                seg->s_szc = 0;
                return (0);
        }

        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                ASSERT(svd->amp == NULL);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                    HAT_REGION_TEXT);
                svd->rcookie = HAT_INVALID_REGION_COOKIE;
        } else if (svd->tr_state == SEGVN_TR_ON) {
                ASSERT(svd->amp != NULL);
                segvn_textunrepl(seg, 1);
                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
                amp = NULL;
        } else {
                if (svd->tr_state != SEGVN_TR_OFF) {
                        ASSERT(svd->tr_state == SEGVN_TR_INIT);
                        svd->tr_state = SEGVN_TR_OFF;
                }

                /*
                 * do HAT_UNLOAD_UNMAP since we are changing the pagesize.
                 * unload argument is 0 when we are freeing the segment
                 * and unload was already done.
                 */
                hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
                    HAT_UNLOAD_UNMAP);
        }

        if (amp == NULL || svd->type == MAP_SHARED) {
                seg->s_szc = 0;
                return (0);
        }

        pgsz = page_get_pagesize(seg->s_szc);
        pages = btop(pgsz);

        /*
         * XXX anon rwlock is not really needed because this is a
         * private segment and we are writers.
         */
        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);

        for (; a < ea; a += pgsz, an_idx += pages) {
                if ((oldap = anon_get_ptr(amp->ahp, an_idx)) != NULL) {
                        ASSERT(vpage != NULL || svd->pageprot == 0);
                        if (vpage != NULL) {
                                ASSERT(sameprot(seg, a, pgsz));
                                prot = VPP_PROT(vpage);
                                pageflag = VPP_ISPPLOCK(vpage) ? LOCK_PAGE : 0;
```

```
                        }
                        if (seg->s_szc != 0) {
                                ASSERT(vp == NULL || anon_pages(amp->ahp,
                                    an_idx, pages) == pages);
                                if ((err = anon_map_demotepages(amp, an_idx,
                                    seg, a, prot, vpage, svd->cred)) != 0) {
                                        goto out;
                                }
                        } else {
                                if (oldap->an_refcnt == 1) {
                                        continue;
                                }
                                if ((err = anon_getpage(&oldap, &vpprot,
                                    anon_pl, PAGESIZE, seg, a, S_READ,
                                    svd->cred))) {
                                        goto out;
                                }
                                if ((pp = anon_private(&ap, seg, a, prot,
                                    anon_pl[0], pageflag, svd->cred)) == NULL) {
                                        err = ENOMEM;
                                        goto out;
                                }
                                anon_decref(oldap);
                                (void) anon_set_ptr(amp->ahp, an_idx, ap,
                                    ANON_SLEEP);
                                page_unlock(pp);
                        }
                }
                vpage = (vpage == NULL) ? NULL : vpage + pages;
        }

        amp->a_szc = 0;
        seg->s_szc = 0;
out:
        ANON_LOCK_EXIT(&amp->a_rwlock);
        return (err);
}

static int
segvn_claim_pages(
        struct seg *seg,
        struct vpage *svp,
        u_offset_t off,
        ulong_t anon_idx,
        uint_t prot)
{
        pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
        size_t ppasize = (pgcnt + 1) * sizeof (page_t *);
        page_t  **ppa;
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp = svd->amp;
        struct vpage *evp = svp + pgcnt;
        caddr_t addr = ((uintptr_t)(svp - svd->vpage) << PAGESHIFT)
            + seg->s_base;
        struct anon *ap;
        struct vnode *vp = svd->vp;
        page_t *pp;
        pgcnt_t pg_idx, i;
        int err = 0;
        anoff_t aoff;
        int anon = (amp != NULL) ? 1 : 0;

        ASSERT(svd->type == MAP_PRIVATE);
        ASSERT(svd->vpage != NULL);
        ASSERT(seg->s_szc != 0);
        ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
        ASSERT(amp == NULL || IS_P2ALIGNED(anon_idx, pgcnt));
        ASSERT(sameprot(seg, addr, pgcnt << PAGESHIFT));

        if (VPP_PROT(svp) == prot)
                return (1);
        if (!((VPP_PROT(svp) ^ prot) & PROT_WRITE))
                return (1);

        ppa = kmem_alloc(ppasize, KM_SLEEP);
        if (anon && vp != NULL) {
                if (anon_get_ptr(amp->ahp, anon_idx) == NULL) {
                        anon = 0;
                        ASSERT(!anon_pages(amp->ahp, anon_idx, pgcnt));
                }
                ASSERT(!anon ||
                    anon_pages(amp->ahp, anon_idx, pgcnt) == pgcnt);
        }

        for (*ppa = NULL, pg_idx = 0; svp < evp; svp++, anon_idx++) {
                if (!VPP_ISPPLOCK(svp))
                        continue;
                if (anon) {
                        ap = anon_get_ptr(amp->ahp, anon_idx);
                        if (ap == NULL) {
                                panic("segvn_claim_pages: no anon slot");
                        }
                        swap_xlate(ap, &vp, &aoff);
                        off = (u_offset_t)aoff;
                }
                ASSERT(vp != NULL);
                if ((pp = page_lookup(vp,
                    (u_offset_t)off, SE_SHARED)) == NULL) {
                        panic("segvn_claim_pages: no page");
                }
                ppa[pg_idx++] = pp;
                off += PAGESIZE;
        }

        if (ppa[0] == NULL) {
                kmem_free(ppa, ppasize);
                return (1);
        }

        ASSERT(pg_idx <= pgcnt);
        ppa[pg_idx] = NULL;


        /* Find each large page within ppa, and adjust its claim */

        /* Does ppa cover a single large page? */
        if (ppa[0]->p_szc == seg->s_szc) {
                if (prot & PROT_WRITE)
                        err = page_addclaim_pages(ppa);
                else
                        err = page_subclaim_pages(ppa);
        } else {
                for (i = 0; ppa[i]; i += pgcnt) {
                        ASSERT(IS_P2ALIGNED(page_pptonum(ppa[i]), pgcnt));
                        if (prot & PROT_WRITE)
                                err = page_addclaim_pages(&ppa[i]);
                        else
                                err = page_subclaim_pages(&ppa[i]);
                        if (err == 0)
                                break;
                }
        }
```

```

        for (i = 0; i < pg_idx; i++) {
                ASSERT(ppa[i] != NULL);
                page_unlock(ppa[i]);
        }

        kmem_free(ppa, ppasize);
        return (err);
}

/*
 * Returns right (upper address) segment if split occurred.
 * If the address is equal to the beginning or end of its segment it returns
 * the current segment.
 */
static struct seg *
segvn_split_seg(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct seg *nseg;
        size_t nsize;
        struct segvn_data *nsvd;

        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(svd->tr_state == SEGVN_TR_OFF);

        ASSERT(addr >= seg->s_base);
        ASSERT(addr <= seg->s_base + seg->s_size);
        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);

        if (addr == seg->s_base || addr == seg->s_base + seg->s_size)
                return (seg);

        nsize = seg->s_base + seg->s_size - addr;
        seg->s_size = addr - seg->s_base;
        nseg = seg_alloc(seg->s_as, addr, nsize);
        ASSERT(nseg != NULL);
        nseg->s_ops = seg->s_ops;
        nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
        nseg->s_data = (void *)nsvd;
        nseg->s_szc = seg->s_szc;
        *nsvd = *svd;
        ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
        nsvd->seg = nseg;
        rw_init(&nsvd->lock, NULL, RW_DEFAULT, NULL);

        if (nsvd->vp != NULL) {
                VN_HOLD(nsvd->vp);
                nsvd->offset = svd->offset +
                    (uintptr_t)(nseg->s_base - seg->s_base);
                if (nsvd->type == MAP_SHARED)
                        lgrp_shm_policy_init(NULL, nsvd->vp);
        } else {
                /*
                 * The offset for an anonymous segment has no signifigance in
                 * terms of an offset into a file. If we were to use the above
                 * calculation instead, the structures read out of
                 * /proc/<pid>/xmap would be more difficult to decipher since
                 * it would be unclear whether two seemingly contiguous
                 * prxmap_t structures represented different segments or a
                 * single segment that had been split up into multiple prxmap_t
                 * structures (e.g. if some part of the segment had not yet
                 * been faulted in).
                 */
                nsvd->offset = 0;
        }
        ASSERT(svd->softlockcnt == 0);
        ASSERT(svd->softlockcnt_sbase == 0);
        ASSERT(svd->softlockcnt_send == 0);
        crhold(svd->cred);

        if (svd->vpage != NULL) {
                size_t bytes = vpgtob(seg_pages(seg));
                size_t nbytes = vpgtob(seg_pages(nseg));
                struct vpage *ovpage = svd->vpage;

                svd->vpage = kmem_alloc(bytes, KM_SLEEP);
                bcopy(ovpage, svd->vpage, bytes);
                nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);
                bcopy(ovpage + seg_pages(seg), nsvd->vpage, nbytes);
                kmem_free(ovpage, bytes + nbytes);
        }
        if (svd->amp != NULL && svd->type == MAP_PRIVATE) {
                struct anon_map *oamp = svd->amp, *namp;
                struct anon_hdr *nahp;

                ANON_LOCK_ENTER(&oamp->a_rwlock, RW_WRITER);
                ASSERT(oamp->refcnt == 1);
                nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
                (void) anon_copy_ptr(oamp->ahp, svd->anon_index,
                    nahp, 0, btop(seg->s_size), ANON_SLEEP);

                namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
                namp->a_szc = nseg->s_szc;
                (void) anon_copy_ptr(oamp->ahp,
                    svd->anon_index + btop(seg->s_size),
                    namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
                anon_release(oamp->ahp, btop(oamp->size));
                oamp->ahp = nahp;
                oamp->size = seg->s_size;
                svd->anon_index = 0;
                nsvd->amp = namp;
                nsvd->anon_index = 0;
                ANON_LOCK_EXIT(&oamp->a_rwlock);
        } else if (svd->amp != NULL) {
                pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
                ASSERT(svd->amp == nsvd->amp);
                ASSERT(seg->s_szc <= svd->amp->a_szc);
                nsvd->anon_index = svd->anon_index + seg_pages(seg);
                ASSERT(IS_P2ALIGNED(nsvd->anon_index, pgcnt));
                ANON_LOCK_ENTER(&svd->amp->a_rwlock, RW_WRITER);
                svd->amp->refcnt++;
                ANON_LOCK_EXIT(&svd->amp->a_rwlock);
        }

        /*
         * Split the amount of swap reserved.
         */
        if (svd->swresv) {
                /*
                 * For MAP_NORESERVE, only allocate swap reserve for pages
                 * being used.  Other segments get enough to cover whole
                 * segment.
                 */
                if (svd->flags & MAP_NORESERVE) {
                        size_t  oswresv;

                        ASSERT(svd->amp);
                        oswresv = svd->swresv;
                        svd->swresv = ptob(anon_pages(svd->amp->ahp,
                            svd->anon_index, btop(seg->s_size)));
```

```
                        nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
                            nsvd->anon_index, btop(nseg->s_size)));
                        ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
                } else {
                        if (svd->pageswap) {
                                svd->swresv = segvn_count_swap_by_vpages(seg);
                                ASSERT(nsvd->swresv >= svd->swresv);
                                nsvd->swresv -= svd->swresv;
                        } else {
                                ASSERT(svd->swresv == seg->s_size +
                                    nseg->s_size);
                                svd->swresv = seg->s_size;
                                nsvd->swresv = nseg->s_size;
                        }
                }
        }

        return (nseg);
}

/*
 * called on memory operations (unmap, setprot, setpagesize) for a subset
 * of a large page segment to either demote the memory range (SDR_RANGE)
 * or the ends (SDR_END) by addr/len.
 *
 * returns 0 on success. returns errno, including ENOMEM, on failure.
 */
static int
segvn_demote_range(
        struct seg *seg,
        caddr_t addr,
        size_t len,
        int flag,
        uint_t szcvec)
{
        caddr_t eaddr = addr + len;
        caddr_t lpgaddr, lpgeaddr;
        struct seg *nseg;
        struct seg *badseg1 = NULL;
        struct seg *badseg2 = NULL;
        size_t pgsz;
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        int err;
        uint_t szc = seg->s_szc;
        uint_t tszcvec;

        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(svd->tr_state == SEGVN_TR_OFF);
        ASSERT(szc != 0);
        pgsz = page_get_pagesize(szc);
        ASSERT(seg->s_base != addr || seg->s_size != len);
        ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);
        ASSERT(svd->softlockcnt == 0);
        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
        ASSERT(szcvec == 0 || (flag == SDR_END && svd->type == MAP_SHARED));

        CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
        ASSERT(flag == SDR_RANGE || eaddr < lpgeaddr || addr > lpgaddr);
        if (flag == SDR_RANGE) {
                /* demote entire range */
                badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
                (void) segvn_split_seg(nseg, lpgeaddr);
                ASSERT(badseg1->s_base == lpgaddr);
                ASSERT(badseg1->s_size == lpgeaddr - lpgaddr);
        } else if (addr != lpgaddr) {
                ASSERT(flag == SDR_END);
                badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
                if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz &&
                    eaddr < lpgaddr + 2 * pgsz) {
                        (void) segvn_split_seg(nseg, lpgeaddr);
                        ASSERT(badseg1->s_base == lpgaddr);
                        ASSERT(badseg1->s_size == 2 * pgsz);
                } else {
                        nseg = segvn_split_seg(nseg, lpgaddr + pgsz);
                        ASSERT(badseg1->s_base == lpgaddr);
                        ASSERT(badseg1->s_size == pgsz);
                        if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz) {
                                ASSERT(lpgeaddr - lpgaddr > 2 * pgsz);
                                nseg = segvn_split_seg(nseg, lpgeaddr - pgsz);
                                badseg2 = nseg;
                                (void) segvn_split_seg(nseg, lpgeaddr);
                                ASSERT(badseg2->s_base == lpgeaddr - pgsz);
                                ASSERT(badseg2->s_size == pgsz);
                        }
                }
        } else {
                ASSERT(flag == SDR_END);
                ASSERT(eaddr < lpgeaddr);
                badseg1 = nseg = segvn_split_seg(seg, lpgeaddr - pgsz);
                (void) segvn_split_seg(nseg, lpgeaddr);
                ASSERT(badseg1->s_base == lpgeaddr - pgsz);
                ASSERT(badseg1->s_size == pgsz);
        }

        ASSERT(badseg1 != NULL);
        ASSERT(badseg1->s_szc == szc);
        ASSERT(flag == SDR_RANGE || badseg1->s_size == pgsz ||
            badseg1->s_size == 2 * pgsz);
        ASSERT(sameprot(badseg1, badseg1->s_base, pgsz));
        ASSERT(badseg1->s_size == pgsz ||
            sameprot(badseg1, badseg1->s_base + pgsz, pgsz));
        if (err = segvn_clrszc(badseg1)) {
                return (err);
        }
        ASSERT(badseg1->s_szc == 0);

        if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
                uint_t tszc = highbit(tszcvec) - 1;
                caddr_t ta = MAX(addr, badseg1->s_base);
                caddr_t te;
                size_t tpgsz = page_get_pagesize(tszc);

                ASSERT(svd->type == MAP_SHARED);
                ASSERT(flag == SDR_END);
                ASSERT(tszc < szc && tszc > 0);

                if (eaddr > badseg1->s_base + badseg1->s_size) {
                        te = badseg1->s_base + badseg1->s_size;
                } else {
                        te = eaddr;
                }

                ASSERT(ta <= te);
                badseg1->s_szc = tszc;
                if (!IS_P2ALIGNED(ta, tpgsz) || !IS_P2ALIGNED(te, tpgsz)) {
                        if (badseg2 != NULL) {
                                err = segvn_demote_range(badseg1, ta, te - ta,
                                    SDR_END, tszcvec);
                                if (err != 0) {
                                        return (err);
                                }
                        } else {
```

```
                                return (segvn_demote_range(badseg1, ta,
                                    te - ta, SDR_END, tszcvec));
                        }
                }
        }

        if (badseg2 == NULL)
                return (0);
        ASSERT(badseg2->s_szc == szc);
        ASSERT(badseg2->s_size == pgsz);
        ASSERT(sameprot(badseg2, badseg2->s_base, badseg2->s_size));
        if (err = segvn_clrszc(badseg2)) {
                return (err);
        }
        ASSERT(badseg2->s_szc == 0);

        if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
                uint_t tszc = highbit(tszcvec) - 1;
                size_t tpgsz = page_get_pagesize(tszc);

                ASSERT(svd->type == MAP_SHARED);
                ASSERT(flag == SDR_END);
                ASSERT(tszc < szc && tszc > 0);
                ASSERT(badseg2->s_base > addr);
                ASSERT(eaddr > badseg2->s_base);
                ASSERT(eaddr < badseg2->s_base + badseg2->s_size);

                badseg2->s_szc = tszc;
                if (!IS_P2ALIGNED(eaddr, tpgsz)) {
                        return (segvn_demote_range(badseg2, badseg2->s_base,
                            eaddr - badseg2->s_base, SDR_END, tszcvec));
                }
        }

        return (0);
}

static int
segvn_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vp, *evp;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        /*
         * If segment protection can be used, simply check against them.
         */
        if (svd->pageprot == 0) {
                int err;

                err = ((svd->prot & prot) != prot) ? EACCES : 0;
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (err);
        }

        /*
         * Have to check down to the vpage level.
         */
        evp = &svd->vpage[seg_page(seg, addr + len)];
        for (vp = &svd->vpage[seg_page(seg, addr)]; vp < evp; vp++) {
                if ((VPP_PROT(vp) & prot) != prot) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EACCES);
                }
        }
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        return (0);
}

static int
segvn_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        size_t pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        if (pgno != 0) {
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
                if (svd->pageprot == 0) {
                        do {
                                protv[--pgno] = svd->prot;
                        } while (pgno != 0);
                } else {
                        size_t pgoff = seg_page(seg, addr);

                        do {
                                pgno--;
                                protv[pgno] = VPP_PROT(&svd->vpage[pgno+pgoff]);
                        } while (pgno != 0);
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        }
        return (0);
}

static u_offset_t
segvn_getoffset(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        return (svd->offset + (uintptr_t)(addr - seg->s_base));
}

/*ARGSUSED*/
static int
segvn_gettype(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        return (svd->type | (svd->flags & (MAP_NORESERVE | MAP_TEXT |
            MAP_INITDATA)));
}

/*ARGSUSED*/
static int
segvn_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        *vpp = svd->vp;
        return (0);
}
```

```
/*
 * Check to see if it makes sense to do kluster/read ahead to
 * addr + delta relative to the mapping at addr.  We assume here
 * that delta is a signed PAGESIZE'd multiple (which can be negative).
 *
 * For segvn, we currently "approve" of the action if we are
 * still in the segment and it maps from the same vp/off,
 * or if the advice stored in segvn_data or vpages allows it.
 * Currently, klustering is not allowed only if MADV_RANDOM is set.
 */
static int
segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon *oap, *ap;
        ssize_t pd;
        size_t page;
        struct vnode *vp1, *vp2;
        u_offset_t off1, off2;
        struct anon_map *amp;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
            SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

        if (addr + delta < seg->s_base ||
            addr + delta >= (seg->s_base + seg->s_size))
                return (-1);            /* exceeded segment bounds */

        pd = delta / (ssize_t)PAGESIZE; /* divide to preserve sign bit */
        page = seg_page(seg, addr);

        /*
         * Check to see if either of the pages addr or addr + delta
         * have advice set that prevents klustering (if MADV_RANDOM advice
         * is set for entire segment, or MADV_SEQUENTIAL is set and delta
         * is negative).
         */
        if (svd->advice == MADV_RANDOM ||
            svd->advice == MADV_SEQUENTIAL && delta < 0)
                return (-1);
        else if (svd->pageadvice && svd->vpage) {
                struct vpage *bvpp, *evpp;

                bvpp = &svd->vpage[page];
                evpp = &svd->vpage[page + pd];
                if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
                    VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
                        return (-1);
                if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
                    VPP_ADVICE(evpp) == MADV_RANDOM)
                        return (-1);
        }

        if (svd->type == MAP_SHARED)
                return (0);             /* shared mapping - all ok */

        if ((amp = svd->amp) == NULL)
                return (0);             /* off original vnode */

        page += svd->anon_index;

        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

        oap = anon_get_ptr(amp->ahp, page);
        ap = anon_get_ptr(amp->ahp, page + pd);
        ANON_LOCK_EXIT(&amp->a_rwlock);

        if ((oap == NULL && ap != NULL) || (oap != NULL && ap == NULL)) {
                return (-1);            /* one with and one without an anon */
        }

        if (oap == NULL) {              /* implies that ap == NULL */
                return (0);             /* off original vnode */
        }

        /*
         * Now we know we have two anon pointers - check to
         * see if they happen to be properly allocated.
         */

        /*
         * XXX We cheat here and don’t lock the anon slots. We can’t because
         * we may have been called from the anon layer which might already
         * have locked them. We are holding a refcnt on the slots so they
         * can’t disappear. The worst that will happen is we’ll get the wrong
         * names (vp, off) for the slots and make a poor klustering decision.
         */
        swap_xlate(ap, &vp1, &off1);
        swap_xlate(oap, &vp2, &off2);


        if (!VOP_CMP(vp1, vp2, NULL) || off1 - off2 != delta)
                return (-1);
        return (0);
}

/*
 * Swap the pages of seg out to secondary storage, returning the
 * number of bytes of storage freed.
 *
 * The basic idea is first to unload all translations and then to call
 * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
 * swap device.  Pages to which other segments have mappings will remain
 * mapped and won’t be swapped.  Our caller (as_swapout) has already
 * performed the unloading step.
 *
 * The value returned is intended to correlate well with the process’s
 * memory requirements.  However, there are some caveats:
 * 1)   When given a shared segment as argument, this routine will
 *      only succeed in swapping out pages for the last sharer of the
 *      segment.  (Previous callers will only have decremented mapping
 *      reference counts.)
 * 2)   We assume that the hat layer maintains a large enough translation
 *      cache to capture process reference patterns.
 */
static size_t
segvn_swapout(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp;
        pgcnt_t pgcnt = 0;
        pgcnt_t npages;
        pgcnt_t page;
        ulong_t anon_index;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        /*
         * Find pages unmapped by our caller and force them
```

```
         * out to the virtual swap device.
         */
        if ((amp = svd->amp) != NULL)
                anon_index = svd->anon_index;
        npages = seg->s_size >> PAGESHIFT;
        for (page = 0; page < npages; page++) {
                page_t *pp;
                struct anon *ap;
                struct vnode *vp;
                u_offset_t off;
                anon_sync_obj_t cookie;

                /*
                 * Obtain <vp, off> pair for the page, then look it up.
                 *
                 * Note that this code is willing to consider regular
                 * pages as well as anon pages.  Is this appropriate here?
                 */
                ap = NULL;
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        if (anon_array_try_enter(amp, anon_index + page,
                            &cookie)) {
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                continue;
                        }
                        ap = anon_get_ptr(amp->ahp, anon_index + page);
                        if (ap != NULL) {
                                swap_xlate(ap, &vp, &off);
                        } else {
                                vp = svd->vp;
                                off = svd->offset + ptob(page);
                        }
                        anon_array_exit(&cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
                        vp = svd->vp;
                        off = svd->offset + ptob(page);
                }
                if (vp == NULL) {               /* untouched zfod page */
                        ASSERT(ap == NULL);
                        continue;
                }

                pp = page_lookup_nowait(vp, off, SE_SHARED);
                if (pp == NULL)
                        continue;


                /*
                 * Examine the page to see whether it can be tossed out,
                 * keeping track of how many we’ve found.
                 */
                if (!page_tryupgrade(pp)) {
                        /*
                         * If the page has an i/o lock and no mappings,
                         * it’s very likely that the page is being
                         * written out as a result of klustering.
                         * Assume this is so and take credit for it here.
                         */
                        if (!page_io_trylock(pp)) {
                                if (!hat_page_is_mapped(pp))
                                        pgcnt++;
                        } else {
                                page_io_unlock(pp);
                        }
                        page_unlock(pp);
                        continue;
                }
                ASSERT(!page_iolock_assert(pp));


                /*
                 * Skip if page is locked or has mappings.
                 * We don’t need the page_struct_lock to look at lckcnt
                 * and cowcnt because the page is exclusive locked.
                 */
                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0 ||
                    hat_page_is_mapped(pp)) {
                        page_unlock(pp);
                        continue;
                }

                /*
                 * dispose skips large pages so try to demote first.
                 */
                if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
                        page_unlock(pp);
                        /*
                         * XXX should skip the remaining page_t’s of this
                         * large page.
                         */
                        continue;
                }

                ASSERT(pp->p_szc == 0);

                /*
                 * No longer mapped -- we can toss it out.  How
                 * we do so depends on whether or not it’s dirty.
                 */
                if (hat_ismod(pp) && pp->p_vnode) {
                        /*
                         * We must clean the page before it can be
                         * freed.  Setting B_FREE will cause pvn_done
                         * to free the page when the i/o completes.
                         * XXX: This also causes it to be accounted
                         *      as a pageout instead of a swap: need
                         *      B_SWAPOUT bit to use instead of B_FREE.
                         *
                         * Hold the vnode before releasing the page lock
                         * to prevent it from being freed and re-used by
                         * some other thread.
                         */
                        VN_HOLD(vp);
                        page_unlock(pp);

                        /*
                         * Queue all i/o requests for the pageout thread
                         * to avoid saturating the pageout devices.
                         */
                        if (!queue_io_request(vp, off))
                                VN_RELE(vp);
                } else {
                        /*
                         * The page was clean, free it.
                         *
                         * XXX: Can we ever encounter modified pages
                         *      with no associated vnode here?
                         */
                        ASSERT(pp->p_vnode != NULL);
                        /*LINTED: constant in conditional context*/
```

```
                        VN_DISPOSE(pp, B_FREE, 0, kcred);
                }

                /*
                 * Credit now even if i/o is in progress.
                 */
                pgcnt++;
        }
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

        /*
         * Wakeup pageout to initiate i/o on all queued requests.
         */
        cv_signal_pageout();
        return (ptob(pgcnt));
}

/*
 * Synchronize primary storage cache with real object in virtual memory.
 *
 * XXX - Anonymous pages should not be sync’ed out at all.
 */
static int
segvn_sync(struct seg *seg, caddr_t addr, size_t len, int attr, uint_t flags)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vpp;
        page_t *pp;
        u_offset_t offset;
        struct vnode *vp;
        u_offset_t off;
        caddr_t eaddr;
        int bflags;
        int err = 0;
        int segtype;
        int pageprot;
        int prot;
        ulong_t anon_index;
        struct anon_map *amp;
        struct anon *ap;
        anon_sync_obj_t cookie;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

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

        if (svd->softlockcnt > 0) {
                /*
                 * If this is shared segment non 0 softlockcnt
                 * means locked pages are still in use.
                 */
                if (svd->type == MAP_SHARED) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EAGAIN);
                }

                /*
                 * flush all pages from seg cache
                 * otherwise we may deadlock in swap_putpage
                 * for B_INVAL page (4175402).
                 *
                 * Even if we grab segvn WRITER’s lock
                 * here, there might be another thread which could’ve
                 * successfully performed lookup/insert just before
                 * we acquired the lock here.  So, grabbing either
                 * lock here is of not much use.  Until we devise
                 * a strategy at upper layers to solve the
                 * synchronization issues completely, we expect
                 * applications to handle this appropriately.
                 */
                segvn_purge(seg);
                if (svd->softlockcnt > 0) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EAGAIN);
                }
        } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
            svd->amp->a_softlockcnt > 0) {
                /*
                 * Try to purge this amp’s entries from pcache. It will
                 * succeed only if other segments that share the amp have no
                 * outstanding softlock’s.
                 */
                segvn_purge(seg);
                if (svd->amp->a_softlockcnt > 0 || svd->softlockcnt > 0) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (EAGAIN);
                }
        }

        vpp = svd->vpage;
        offset = svd->offset + (uintptr_t)(addr - seg->s_base);
        bflags = ((flags & MS_ASYNC) ? B_ASYNC : 0) |
            ((flags & MS_INVALIDATE) ? B_INVAL : 0);

        if (attr) {
                pageprot = attr & ~(SHARED|PRIVATE);
                segtype = (attr & SHARED) ? MAP_SHARED : MAP_PRIVATE;

                /*
                 * We are done if the segment types don’t match
                 * or if we have segment level protections and
                 * they don’t match.
                 */
                if (svd->type != segtype) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
                if (vpp == NULL) {
                        if (svd->prot != pageprot) {
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                return (0);
                        }
                        prot = svd->prot;
                } else
                        vpp = &svd->vpage[seg_page(seg, addr)];

        } else if (svd->vp && svd->amp == NULL &&
            (flags & MS_INVALIDATE) == 0) {

                /*
                 * No attributes, no anonymous pages and MS_INVALIDATE flag
                 * is not on, just use one big request.
                 */
                err = VOP_PUTPAGE(svd->vp, (offset_t)offset, len,
                    bflags, svd->cred, NULL);
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (err);
        }

        if ((amp = svd->amp) != NULL)
                anon_index = svd->anon_index + seg_page(seg, addr);
```

```
        for (eaddr = addr + len; addr < eaddr; addr += PAGESIZE) {
                ap = NULL;
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        anon_array_enter(amp, anon_index, &cookie);
                        ap = anon_get_ptr(amp->ahp, anon_index++);
                        if (ap != NULL) {
                                swap_xlate(ap, &vp, &off);
                        } else {
                                vp = svd->vp;
                                off = offset;
                        }
                        anon_array_exit(&cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
                        vp = svd->vp;
                        off = offset;
                }
                offset += PAGESIZE;

                if (vp == NULL)         /* untouched zfod page */
                        continue;

                if (attr) {
                        if (vpp) {
                                prot = VPP_PROT(vpp);
                                vpp++;
                        }
                        if (prot != pageprot) {
                                continue;
                        }
                }

                /*
                 * See if any of these pages are locked --  if so, then we
                 * will have to truncate an invalidate request at the first
                 * locked one. 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
                 * we do the PUTPAGE, then PUTPAGE simply does nothing.
                 */
                if (flags & MS_INVALIDATE) {
                        if ((pp = page_lookup(vp, off, SE_SHARED)) != NULL) {
                                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0) {
                                        page_unlock(pp);
                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                        return (EBUSY);
                                }
                                if (ap != NULL && pp->p_szc != 0 &&
                                    page_tryupgrade(pp)) {
                                        if (pp->p_lckcnt == 0 &&
                                            pp->p_cowcnt == 0) {
                                                /*
                                                 * swapfs VN_DISPOSE() won’t
                                                 * invalidate large pages.
                                                 * Attempt to demote.
                                                 * XXX can’t help it if it
                                                 * fails. But for swapfs
                                                 * pages it is no big deal.
                                                 */
                                                (void) page_try_demote_pages(
                                                    pp);
                                        }
                                }
                                page_unlock(pp);
                        }
                } else if (svd->type == MAP_SHARED && amp != NULL) {
                        /*
                         * Avoid writing out to disk ISM's large pages
                         * because segspt_free_pages() relies on NULL an_pvp
                         * of anon slots of such pages.
                         */

                        ASSERT(svd->vp == NULL);
                        /*
                         * swapfs uses page_lookup_nowait if not freeing or
                         * invalidating and skips a page if
                         * page_lookup_nowait returns NULL.
                         */
                        pp = page_lookup_nowait(vp, off, SE_SHARED);
                        if (pp == NULL) {
                                continue;
                        }
                        if (pp->p_szc != 0) {
                                page_unlock(pp);
                                continue;
                        }

                        /*
                         * Note ISM pages are created large so (vp, off)'s
                         * page cannot suddenly become large after we unlock
                         * pp.
                         */
                        page_unlock(pp);
                }
                /*
                 * XXX - Should ultimately try to kluster
                 * calls to VOP_PUTPAGE() for performance.
                 */
                VN_HOLD(vp);
                err = VOP_PUTPAGE(vp, (offset_t)off, PAGESIZE,
                    (bflags | (IS_SWAPFSVP(vp) ? B_PAGE_NOWAIT : 0)),
                    svd->cred, NULL);

                VN_RELE(vp);
                if (err)
                        break;
        }
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        return (err);
}

/*
 * Determine if we have data corresponding to pages in the
 * primary storage virtual memory cache (i.e., "in core").
 */
static size_t
segvn_incore(struct seg *seg, caddr_t addr, size_t len, char *vec)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vnode *vp, *avp;
        u_offset_t offset, aoffset;
        size_t p, ep;
        int ret;
        struct vpage *vpp;
        page_t *pp;
        uint_t start;
        struct anon_map *amp;           /* XXX - for locknest */
        struct anon *ap;
        uint_t attr;
        anon_sync_obj_t cookie;
```

```
        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        if (svd->amp == NULL && svd->vp == NULL) {
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                bzero(vec, btopr(len));
                return (len);   /* no anonymous pages created yet */
        }

        p = seg_page(seg, addr);
        ep = seg_page(seg, addr + len);
        start = svd->vp ? SEG_PAGE_VNODEBACKED : 0;

        amp = svd->amp;
        for (; p < ep; p++, addr += PAGESIZE) {
                vpp = (svd->vpage) ? &svd->vpage[p]: NULL;
                ret = start;
                ap = NULL;
                avp = NULL;
                /* Grab the vnode/offset for the anon slot */
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        anon_array_enter(amp, svd->anon_index + p, &cookie);
                        ap = anon_get_ptr(amp->ahp, svd->anon_index + p);
                        if (ap != NULL) {
                                swap_xlate(ap, &avp, &aoffset);
                        }
                        anon_array_exit(&cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }
                if ((avp != NULL) && page_exists(avp, aoffset)) {
                        /* A page exists for the anon slot */
                        ret |= SEG_PAGE_INCORE;

                        /*
                         * If page is mapped and writable
                         */
                        attr = (uint_t)0;
                        if ((hat_getattr(seg->s_as->a_hat, addr,
                            &attr) != -1) && (attr & PROT_WRITE)) {
                                ret |= SEG_PAGE_ANON;
                        }
                        /*
                         * Don't get page_struct lock for lckcnt and cowcnt,
                         * since this is purely advisory.
                         */
                        if ((pp = page_lookup_nowait(avp, aoffset,
                            SE_SHARED)) != NULL) {
                                if (pp->p_lckcnt)
                                        ret |= SEG_PAGE_SOFTLOCK;
                                if (pp->p_cowcnt)
                                        ret |= SEG_PAGE_HASCOW;
                                page_unlock(pp);
                        }
                }

                /* Gather vnode statistics */
                vp = svd->vp;
                offset = svd->offset + (uintptr_t)(addr - seg->s_base);

                if (vp != NULL) {
                        /*
                         * Try to obtain a "shared" lock on the page
                         * without blocking.  If this fails, determine
                         * if the page is in memory.
                         */
                        pp = page_lookup_nowait(vp, offset, SE_SHARED);
                        if ((pp == NULL) && (page_exists(vp, offset))) {
                                /* Page is incore, and is named */
                                ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
                        }
                        /*
                         * Don’t get page_struct lock for lckcnt and cowcnt,
                         * since this is purely advisory.
                         */
                        if (pp != NULL) {
                                ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
                                if (pp->p_lckcnt)
                                        ret |= SEG_PAGE_SOFTLOCK;
                                if (pp->p_cowcnt)
                                        ret |= SEG_PAGE_HASCOW;
                                page_unlock(pp);
                        }
                }

                /* Gather virtual page information */
                if (vpp) {
                        if (VPP_ISPPLOCK(vpp))
                                ret |= SEG_PAGE_LOCKED;
                        vpp++;
                }

                *vec++ = (char)ret;
        }
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        return (len);
}

/*
 * Statement for p_cowcnts/p_lckcnts.
 *
 * p_cowcnt is updated while mlock/munlocking MAP_PRIVATE and PROT_WRITE region
 * irrespective of the following factors or anything else:
 *
 *      (1) anon slots are populated or not
 *      (2) cow is broken or not
 *      (3) refcnt on ap is 1 or greater than 1
 *
 * If it’s not MAP_PRIVATE and PROT_WRITE, p_lckcnt is updated during mlock
 * and munlock.
 *
 *
 * Handling p_cowcnts/p_lckcnts during copy-on-write fault:
 *
 *      if vpage has PROT_WRITE
 *              transfer cowcnt on the oldpage -> cowcnt on the newpage
 *      else
 *              transfer lckcnt on the oldpage -> lckcnt on the newpage
 *
 *      During copy-on-write, decrement p_cowcnt on the oldpage and increment
 *      p_cowcnt on the newpage *if* the corresponding vpage has PROT_WRITE.
 *
 *      We may also break COW if softlocking on read access in the physio case.
 *      In this case, vpage may not have PROT_WRITE. So, we need to decrement
 *      p_lckcnt on the oldpage and increment p_lckcnt on the newpage *if* the
 *      vpage doesn’t have PROT_WRITE.
 *
 *
 * Handling p_cowcnts/p_lckcnts during mprotect on mlocked region:
 *
 *      If a MAP_PRIVATE region loses PROT_WRITE, we decrement p_cowcnt and
 *      increment p_lckcnt by calling page_subclaim() which takes care of
```

```
 *      availrmem accounting and p_lckcnt overflow.
 *
 *      If a MAP_PRIVATE region gains PROT_WRITE, we decrement p_lckcnt and
 *      increment p_cowcnt by calling page_addclaim() which takes care of
 *      availrmem availability and p_cowcnt overflow.
 */

/*
 * Lock down (or unlock) pages mapped by this segment.
 *
 * XXX only creates PAGESIZE pages if anon slots are not initialized.
 * At fault time they will be relocated into larger pages.
 */
static int
segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
    int attr, int op, ulong_t *lockmap, size_t pos)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vpp;
        struct vpage *evp;
        page_t *pp;
        u_offset_t offset;
        u_offset_t off;
        int segtype;
        int pageprot;
        int claim;
        struct vnode *vp;
        ulong_t anon_index;
        struct anon_map *amp;
        struct anon *ap;
        struct vattr va;
        anon_sync_obj_t cookie;
        struct kshmid *sp = NULL;
        struct proc     *p = curproc;
        kproject_t      *proj = NULL;
        int chargeproc = 1;
        size_t locked_bytes = 0;
        size_t unlocked_bytes = 0;
        int err = 0;

        /*
         * Hold write lock on address space because may split or concatenate
         * segments
         */
        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        /*
         * If this is a shm, use shm’s project and zone, else use
         * project and zone of calling process
         */

        /* Determine if this segment backs a sysV shm */
        if (svd->amp != NULL && svd->amp->a_sp != NULL) {
                ASSERT(svd->type == MAP_SHARED);
                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                sp = svd->amp->a_sp;
                proj = sp->shm_perm.ipc_proj;
                chargeproc = 0;
        }

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
        if (attr) {
                pageprot = attr & ~(SHARED|PRIVATE);
                segtype = attr & SHARED ? MAP_SHARED : MAP_PRIVATE;

                /*
```

```
                 * We are done if the segment types don't match
                 * or if we have segment level protections and
                 * they don't match.
                 */
                if (svd->type != segtype) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
                if (svd->pageprot == 0 && svd->prot != pageprot) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
        }

        if (op == MC_LOCK) {
                if (svd->tr_state == SEGVN_TR_INIT) {
                        svd->tr_state = SEGVN_TR_OFF;
                } else if (svd->tr_state == SEGVN_TR_ON) {
                        ASSERT(svd->amp != NULL);
                        segvn_textunrepl(seg, 0);
                        ASSERT(svd->amp == NULL &&
                            svd->tr_state == SEGVN_TR_OFF);
                }
        }

        /*
         * If we're locking, then we must create a vpage structure if
         * none exists.  If we're unlocking, then check to see if there
         * is a vpage --  if not, then we could not have locked anything.
         */

        if ((vpp = svd->vpage) == NULL) {
                if (op == MC_LOCK)
                        segvn_vpage(seg);
                else {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
        }

        /*
         * The anonymous data vector (i.e., previously
         * unreferenced mapping to swap space) can be allocated
         * by lazily testing for its existence.
         */
        if (op == MC_LOCK && svd->amp == NULL && svd->vp == NULL) {
                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
                svd->amp->a_szc = seg->s_szc;
        }

        if ((amp = svd->amp) != NULL) {
                anon_index = svd->anon_index + seg_page(seg, addr);
        }

        offset = svd->offset + (uintptr_t)(addr - seg->s_base);
        evp = &svd->vpage[seg_page(seg, addr + len)];

        if (sp != NULL)
                mutex_enter(&sp->shm_mlock);

        /* determine number of unlocked bytes in range for lock operation */
        if (op == MC_LOCK) {

                if (sp == NULL) {
                        for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
```

```
                            vpp++) {
                                if (!VPP_ISPPLOCK(vpp))
                                        unlocked_bytes += PAGESIZE;
                        }
                } else {
                        ulong_t         i_idx, i_edx;
                        anon_sync_obj_t i_cookie;
                        struct anon     *i_ap;
                        struct vnode    *i_vp;
                        u_offset_t      i_off;

                        /* Only count sysV pages once for locked memory */
                        i_edx = svd->anon_index + seg_page(seg, addr + len);
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        for (i_idx = anon_index; i_idx < i_edx; i_idx++) {
                                anon_array_enter(amp, i_idx, &i_cookie);
                                i_ap = anon_get_ptr(amp->ahp, i_idx);
                                if (i_ap == NULL) {
                                        unlocked_bytes += PAGESIZE;
                                        anon_array_exit(&i_cookie);
                                        continue;
                                }
                                swap_xlate(i_ap, &i_vp, &i_off);
                                anon_array_exit(&i_cookie);
                                pp = page_lookup(i_vp, i_off, SE_SHARED);
                                if (pp == NULL) {
                                        unlocked_bytes += PAGESIZE;
                                        continue;
                                } else if (pp->p_lckcnt == 0)
                                        unlocked_bytes += PAGESIZE;
                                page_unlock(pp);
                        }
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                }

                mutex_enter(&p->p_lock);
                err = rctl_incr_locked_mem(p, proj, unlocked_bytes,
                    chargeproc);
                mutex_exit(&p->p_lock);

                if (err) {
                        if (sp != NULL)
                                mutex_exit(&sp->shm_mlock);
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (err);
                }
        }
        /*
         * Loop over all pages in the range.  Process if we're locking and
         * page has not already been locked in this mapping; or if we're
         * unlocking and the page has been locked.
         */
        for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
            vpp++, pos++, addr += PAGESIZE, offset += PAGESIZE, anon_index++) {
                if ((attr == 0 || VPP_PROT(vpp) == pageprot) &&
                    ((op == MC_LOCK && !VPP_ISPPLOCK(vpp)) ||
                    (op == MC_UNLOCK && VPP_ISPPLOCK(vpp)))) {

                        if (amp != NULL)
                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        /*
                         * If this isn't a MAP_NORESERVE segment and
                         * we're locking, allocate anon slots if they
                         * don't exist.  The page is brought in later on.
                         */
                        if (op == MC_LOCK && svd->vp == NULL &&
                            ((svd->flags & MAP_NORESERVE) == 0) &&
                            amp != NULL &&
                            ((ap = anon_get_ptr(amp->ahp, anon_index))
                            == NULL)) {
                                anon_array_enter(amp, anon_index, &cookie);

                                if ((ap = anon_get_ptr(amp->ahp,
                                    anon_index)) == NULL) {
                                        pp = anon_zero(seg, addr, &ap,
                                            svd->cred);
                                        if (pp == NULL) {
                                                anon_array_exit(&cookie);
                                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                                err = ENOMEM;
                                                goto out;
                                        }
                                        ASSERT(anon_get_ptr(amp->ahp,
                                            anon_index) == NULL);
                                        (void) anon_set_ptr(amp->ahp,
                                            anon_index, ap, ANON_SLEEP);
                                        page_unlock(pp);
                                }
                                anon_array_exit(&cookie);
                        }

                        /*
                         * Get name for page, accounting for
                         * existence of private copy.
                         */
                        ap = NULL;
                        if (amp != NULL) {
                                anon_array_enter(amp, anon_index, &cookie);
                                ap = anon_get_ptr(amp->ahp, anon_index);
                                if (ap != NULL) {
                                        swap_xlate(ap, &vp, &off);
                                } else {
                                        if (svd->vp == NULL &&
                                            (svd->flags & MAP_NORESERVE)) {
                                                anon_array_exit(&cookie);
                                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                                continue;
                                        }
                                        vp = svd->vp;
                                        off = offset;
                                }
                                if (op != MC_LOCK || ap == NULL) {
                                        anon_array_exit(&cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                        } else {
                                vp = svd->vp;
                                off = offset;
                        }

                        /*
                         * Get page frame.  It’s ok if the page is
                         * not available when we’re unlocking, as this
                         * may simply mean that a page we locked got
                         * truncated out of existence after we locked it.
                         *
                         * Invoke VOP_GETPAGE() to obtain the page struct
                         * since we may need to read it from disk if its
                         * been paged out.
                         */
                        if (op != MC_LOCK)
                                pp = page_lookup(vp, off, SE_SHARED);
```

```
                        else {
                                page_t *pl[1 + 1];
                                int error;

                                ASSERT(vp != NULL);

                                error = VOP_GETPAGE(vp, (offset_t)off, PAGESIZE,
                                    (uint_t *)NULL, pl, PAGESIZE, seg, addr,
                                    S_OTHER, svd->cred, NULL);

                                if (error && ap != NULL) {
                                        anon_array_exit(&cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }

                                /*
                                 * If the error is EDEADLK then we must bounce
                                 * up and drop all vm subsystem locks and then
                                 * retry the operation later
                                 * This behavior is a temporary measure because
                                 * ufs/sds logging is badly designed and will
                                 * deadlock if we don’t allow this bounce to
                                 * happen.  The real solution is to re-design
                                 * the logging code to work properly.  See bug
                                 * 4125102 for details of the problem.
                                 */
                                if (error == EDEADLK) {
                                        err = error;
                                        goto out;
                                }
                                /*
                                 * Quit if we fail to fault in the page.  Treat
                                 * the failure as an error, unless the addr
                                 * is mapped beyond the end of a file.
                                 */
                                if (error && svd->vp) {
                                        va.va_mask = AT_SIZE;
                                        if (VOP_GETATTR(svd->vp, &va, 0,
                                            svd->cred, NULL) != 0) {
                                                err = EIO;
                                                goto out;
                                        }
                                        if (btopr(va.va_size) >=
                                            btopr(off + 1)) {
                                                err = EIO;
                                                goto out;
                                        }
                                        goto out;

                                } else if (error) {
                                        err = EIO;
                                        goto out;
                                }
                                pp = pl[0];
                                ASSERT(pp != NULL);
                        }

                        /*
                         * See Statement at the beginning of this routine.
                         *
                         * claim is always set if MAP_PRIVATE and PROT_WRITE
                         * irrespective of following factors:
                         *
                         * (1) anon slots are populated or not
                         * (2) cow is broken or not
                         * (3) refcnt on ap is 1 or greater than 1
                         *
                         * See 4140683 for details
                         */
                        claim = ((VPP_PROT(vpp) & PROT_WRITE) &&
                            (svd->type == MAP_PRIVATE));

                        /*
                         * Perform page-level operation appropriate to
                         * operation.  If locking, undo the SOFTLOCK
                         * performed to bring the page into memory
                         * after setting the lock.  If unlocking,
                         * and no page was found, account for the claim
                         * separately.
                         */
                        if (op == MC_LOCK) {
                                int ret = 1;    /* Assume success */

                                ASSERT(!VPP_ISPPLOCK(vpp));

                                ret = page_pp_lock(pp, claim, 0);
                                if (ap != NULL) {
                                        if (ap->an_pvp != NULL) {
                                                anon_swap_free(ap, pp);
                                        }
                                        anon_array_exit(&cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                }
                                if (ret == 0) {
                                        /* locking page failed */
                                        page_unlock(pp);
                                        err = EAGAIN;
                                        goto out;
                                }
                                VPP_SETPPLOCK(vpp);
                                if (sp != NULL) {
                                        if (pp->p_lckcnt == 1)
                                                locked_bytes += PAGESIZE;
                                } else
                                        locked_bytes += PAGESIZE;

                                if (lockmap != (ulong_t *)NULL)
                                        BT_SET(lockmap, pos);

                                page_unlock(pp);
                        } else {
                                ASSERT(VPP_ISPPLOCK(vpp));
                                if (pp != NULL) {
                                        /* sysV pages should be locked */
                                        ASSERT(sp == NULL || pp->p_lckcnt > 0);
                                        page_pp_unlock(pp, claim, 0);
                                        if (sp != NULL) {
                                                if (pp->p_lckcnt == 0)
                                                        unlocked_bytes
                                                            += PAGESIZE;
                                        } else
                                                unlocked_bytes += PAGESIZE;
                                        page_unlock(pp);
                                } else {
                                        ASSERT(sp == NULL);
                                        unlocked_bytes += PAGESIZE;
                                }
                                VPP_CLRPPLOCK(vpp);
                        }
                }
        }
out:
```

```
        if (op == MC_LOCK) {
                /* Credit back bytes that did not get locked */
                if ((unlocked_bytes - locked_bytes) > 0) {
                        if (proj == NULL)
                                mutex_enter(&p->p_lock);
                        rctl_decr_locked_mem(p, proj,
                            (unlocked_bytes - locked_bytes), chargeproc);
                        if (proj == NULL)
                                mutex_exit(&p->p_lock);
                }

        } else {
                /* Account bytes that were unlocked */
                if (unlocked_bytes > 0) {
                        if (proj == NULL)
                                mutex_enter(&p->p_lock);
                        rctl_decr_locked_mem(p, proj, unlocked_bytes,
                            chargeproc);
                        if (proj == NULL)
                                mutex_exit(&p->p_lock);
                }
        }
        if (sp != NULL)
                mutex_exit(&sp->shm_mlock);
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

        return (err);
}

/*
 * Set advice from user for specified pages
 * There are 5 types of advice:
 *      MADV_NORMAL     - Normal (default) behavior (whatever that is)
 *      MADV_RANDOM     - Random page references
 *                              do not allow readahead or ’klustering’
 *      MADV_SEQUENTIAL - Sequential page references
 *                              Pages previous to the one currently being
 *                              accessed (determined by fault) are ’not needed’
 *                              and are freed immediately
 *      MADV_WILLNEED   - Pages are likely to be used (fault ahead in mctl)
 *      MADV_DONTNEED   - Pages are not needed (synced out in mctl)
 *      MADV_FREE       - Contents can be discarded
 *      MADV_ACCESS_DEFAULT- Default access
 *      MADV_ACCESS_LWP - Next LWP will access heavily
 *      MADV_ACCESS_MANY- Many LWPs or processes will access heavily
 */
static int
segvn_advise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        size_t page;
        int err = 0;
        int already_set;
        struct anon_map *amp;
        ulong_t anon_index;
        struct seg *next;
        lgrp_mem_policy_t policy;
        struct seg *prev;
        struct vnode *vp;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        /*
         * In case of MADV_FREE, we won’t be modifying any segment private
         * data structures; so, we only need to grab READER’s lock
         */
        if (behav != MADV_FREE) {
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
                if (svd->tr_state != SEGVN_TR_OFF) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }
        } else {
                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        }

        /*
         * Large pages are assumed to be only turned on when accesses to the
         * segment's address range have spatial and temporal locality. That
         * justifies ignoring MADV_SEQUENTIAL for large page segments.
         * Also, ignore advice affecting lgroup memory allocation
         * if don't need to do lgroup optimizations on this system
         */

        if ((behav == MADV_SEQUENTIAL &&
            (seg->s_szc != 0 || HAT_IS_REGION_COOKIE_VALID(svd->rcookie))) ||
            (!lgrp_optimizations() && (behav == MADV_ACCESS_DEFAULT ||
            behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY))) {
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);
        }

        if (behav == MADV_SEQUENTIAL || behav == MADV_ACCESS_DEFAULT ||
            behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY) {
                /*
                 * Since we are going to unload hat mappings
                 * we first have to flush the cache. Otherwise
                 * this might lead to system panic if another
                 * thread is doing physio on the range whose
                 * mappings are unloaded by madvise(3C).
                 */
                if (svd->softlockcnt > 0) {
                        /*
                         * If this is shared segment non 0 softlockcnt
                         * means locked pages are still in use.
                         */
                        if (svd->type == MAP_SHARED) {
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                return (EAGAIN);
                        }
                        /*
                         * Since we do have the segvn writers lock
                         * nobody can fill the cache with entries
                         * belonging to this seg during the purge.
                         * The flush either succeeds or we still
                         * have pending I/Os. In the later case,
                         * madvise(3C) fails.
                         */
                        segvn_purge(seg);
                        if (svd->softlockcnt > 0) {
                                /*
                                 * Since madvise(3C) is advisory and
                                 * it's not part of UNIX98, madvise(3C)
                                 * failure here doesn't cause any hardship.
                                 * Note that we don't block in "as" layer.
                                 */
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                return (EAGAIN);
                        }
                } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
                    svd->amp->a_softlockcnt > 0) {
                        /*
```

```
                         * Try to purge this amp's entries from pcache. It
                         * will succeed only if other segments that share the
                         * amp have no outstanding softlock's.
                         */
                        segvn_purge(seg);
                }
        }

        amp = svd->amp;
        vp = svd->vp;
        if (behav == MADV_FREE) {
                /*
                 * MADV_FREE is not supported for segments with
                 * underlying object; if anonmap is NULL, anon slots
                 * are not yet populated and there is nothing for
                 * us to do. As MADV_FREE is advisory, we don't
                 * return error in either case.
                 */
                if (vp != NULL || amp == NULL) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        return (0);
                }

                segvn_purge(seg);

                page = seg_page(seg, addr);
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                anon_disclaim(amp, svd->anon_index + page, len);
                ANON_LOCK_EXIT(&amp->a_rwlock);
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                return (0);
        }

        /*
         * If advice is to be applied to entire segment,
         * use advice field in seg_data structure
         * otherwise use appropriate vpage entry.
         */
        if ((addr == seg->s_base) && (len == seg->s_size)) {
                switch (behav) {
                case MADV_ACCESS_LWP:
                case MADV_ACCESS_MANY:
                case MADV_ACCESS_DEFAULT:
                        /*
                         * Set memory allocation policy for this segment
                         */
                        policy = lgrp_madv_to_policy(behav, len, svd->type);
                        if (svd->type == MAP_SHARED)
                                already_set = lgrp_shm_policy_set(policy, amp,
                                    svd->anon_index, vp, svd->offset, len);
                        else {
                                /*
                                 * For private memory, need writers lock on
                                 * address space because the segment may be
                                 * split or concatenated when changing policy
                                 */
                                if (AS_READ_HELD(seg->s_as,
                                    &seg->s_as->a_lock)) {
                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                        return (IE_RETRY);
                                }

                                already_set = lgrp_privm_policy_set(policy,
                                    &svd->policy_info, len);
                        }
                        /*
                         * If policy set already and it shouldn't be reapplied,
                         * don't do anything.
                         */
                        if (already_set &&
                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
                                break;

                        /*
                         * Mark any existing pages in given range for
                         * migration
                         */
                        page_mark_migrate(seg, addr, len, amp, svd->anon_index,
                            vp, svd->offset, 1);

                        /*
                         * If same policy set already or this is a shared
                         * memory segment, don't need to try to concatenate
                         * segment with adjacent ones.
                         */
                        if (already_set || svd->type == MAP_SHARED)
                                break;

                        /*
                         * Try to concatenate this segment with previous
                         * one and next one, since we changed policy for
                         * this one and it may be compatible with adjacent
                         * ones now.
                         */
                        prev = AS_SEGPREV(seg->s_as, seg);
                        next = AS_SEGNEXT(seg->s_as, seg);

                        if (next && next->s_ops == &segvn_ops &&
                            addr + len == next->s_base)
                                (void) segvn_concat(seg, next, 1);

                        if (prev && prev->s_ops == &segvn_ops &&
                            addr == prev->s_base + prev->s_size) {
                                /*
                                 * Drop lock for private data of current
                                 * segment before concatenating (deleting) it
                                 * and return IE_REATTACH to tell as_ctl() that
                                 * current segment has changed
                                 */
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                if (!segvn_concat(prev, seg, 1))
                                        err = IE_REATTACH;

                                return (err);
                        }
                        break;

                case MADV_SEQUENTIAL:
                        /*
                         * unloading mapping guarantees
                         * detection in segvn_fault
                         */
                        ASSERT(seg->s_szc == 0);
                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                        hat_unload(seg->s_as->a_hat, addr, len,
                            HAT_UNLOAD);
                        /* FALLTHROUGH */
                case MADV_NORMAL:
                case MADV_RANDOM:
                        svd->advice = (uchar_t)behav;
                        svd->pageadvice = 0;
```

```
                        break;
                case MADV_WILLNEED:     /* handled in memcntl */
                case MADV_DONTNEED:     /* handled in memcntl */
                case MADV_FREE:         /* handled above */
                        break;
                default:
                        err = EINVAL;
                }
        } else {
                caddr_t                 eaddr;
                struct seg              *new_seg;
                struct segvn_data       *new_svd;
                u_offset_t              off;
                caddr_t                 oldeaddr;

                page = seg_page(seg, addr);

                segvn_vpage(seg);

                switch (behav) {
                        struct vpage *bvpp, *evpp;

                case MADV_ACCESS_LWP:
                case MADV_ACCESS_MANY:
                case MADV_ACCESS_DEFAULT:
                        /*
                         * Set memory allocation policy for portion of this
                         * segment
                         */

                        /*
                         * Align address and length of advice to page
                         * boundaries for large pages
                         */
                        if (seg->s_szc != 0) {
                                size_t  pgsz;

                                pgsz = page_get_pagesize(seg->s_szc);
                                addr = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
                                len = P2ROUNDUP(len, pgsz);
                        }

                        /*
                         * Check to see whether policy is set already
                         */
                        policy = lgrp_madv_to_policy(behav, len, svd->type);

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

                        if (svd->type == MAP_SHARED)
                                already_set = lgrp_shm_policy_set(policy, amp,
                                    anon_index, vp, off, len);
                        else
                                already_set =
                                    (policy == svd->policy_info.mem_policy);

                        /*
                         * If policy set already and it shouldn't be reapplied,
                         * don't do anything.
                         */
                        if (already_set &&
                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
                                break;

                        /*
```

```
                         * For private memory, need writers lock on
                         * address space because the segment may be
                         * split or concatenated when changing policy
                         */
                        if (svd->type == MAP_PRIVATE &&
                            AS_READ_HELD(seg->s_as, &seg->s_as->a_lock)) {
                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                                return (IE_RETRY);
                        }

                        /*
                         * Mark any existing pages in given range for
                         * migration
                         */
                        page_mark_migrate(seg, addr, len, amp, svd->anon_index,
                            vp, svd->offset, 1);

                        /*
                         * Don’t need to try to split or concatenate
                         * segments, since policy is same or this is a shared
                         * memory segment
                         */
                        if (already_set || svd->type == MAP_SHARED)
                                break;

                        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
                                ASSERT(svd->amp == NULL);
                                ASSERT(svd->tr_state == SEGVN_TR_OFF);
                                ASSERT(svd->softlockcnt == 0);
                                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
                                    HAT_REGION_TEXT);
                                svd->rcookie = HAT_INVALID_REGION_COOKIE;
                        }

                        /*
                         * Split off new segment if advice only applies to a
                         * portion of existing segment starting in middle
                         */
                        new_seg = NULL;
                        eaddr = addr + len;
                        oldeaddr = seg->s_base + seg->s_size;
                        if (addr > seg->s_base) {
                                /*
                                 * Must flush I/O page cache
                                 * before splitting segment
                                 */
                                if (svd->softlockcnt > 0)
                                        segvn_purge(seg);

                                /*
                                 * Split segment and return IE_REATTACH to tell
                                 * as_ctl() that current segment changed
                                 */
                                new_seg = segvn_split_seg(seg, addr);
                                new_svd = (struct segvn_data *)new_seg->s_data;
                                err = IE_REATTACH;

                                /*
                                 * If new segment ends where old one
                                 * did, try to concatenate the new
                                 * segment with next one.
                                 */
                                if (eaddr == oldeaddr) {
                                        /*
                                         * Set policy for new segment
                                         */
```

```
                                        (void) lgrp_privm_policy_set(policy,
                                            &new_svd->policy_info,
                                            new_seg->s_size);

                                        next = AS_SEGNEXT(new_seg->s_as,
                                            new_seg);

                                        if (next &&
                                            next->s_ops == &segvn_ops &&
                                            eaddr == next->s_base)
                                                (void) segvn_concat(new_seg,
                                                    next, 1);
                                }
                        }

                        /*
                         * Split off end of existing segment if advice only
                         * applies to a portion of segment ending before
                         * end of the existing segment
                         */
                        if (eaddr < oldeaddr) {
                                /*
                                 * Must flush I/O page cache
                                 * before splitting segment
                                 */
                                if (svd->softlockcnt > 0)
                                        segvn_purge(seg);

                                /*
                                 * If beginning of old segment was already
                                 * split off, use new segment to split end off
                                 * from.
                                 */
                                if (new_seg != NULL && new_seg != seg) {
                                        /*
                                         * Split segment
                                         */
                                        (void) segvn_split_seg(new_seg, eaddr);

                                        /*
                                         * Set policy for new segment
                                         */
                                        (void) lgrp_privm_policy_set(policy,
                                            &new_svd->policy_info,
                                            new_seg->s_size);
                                } else {
                                        /*
                                         * Split segment and return IE_REATTACH
                                         * to tell as_ctl() that current
                                         * segment changed
                                         */
                                        (void) segvn_split_seg(seg, eaddr);
                                        err = IE_REATTACH;

                                        (void) lgrp_privm_policy_set(policy,
                                            &svd->policy_info, seg->s_size);

                                        /*
                                         * If new segment starts where old one
                                         * did, try to concatenate it with
                                         * previous segment.
                                         */
                                        if (addr == seg->s_base) {
                                                prev = AS_SEGPREV(seg->s_as,
                                                    seg);
                                                /*
                                                 * Drop lock for private data
                                                 * of current segment before
                                                 * concatenating (deleting) it
                                                 */
                                                if (prev &&
                                                    prev->s_ops ==
                                                    &segvn_ops &&
                                                    addr == prev->s_base +
                                                    prev->s_size) {
                                                        SEGVN_LOCK_EXIT(
                                                            seg->s_as,
                                                            &svd->lock);
                                                        (void) segvn_concat(
                                                            prev, seg, 1);
                                                        return (err);
                                                }
                                        }
                                }
                        }
                        break;
                case MADV_SEQUENTIAL:
                        ASSERT(seg->s_szc == 0);
                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
                        hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
                        /* FALLTHROUGH */
                case MADV_NORMAL:
                case MADV_RANDOM:
                        bvpp = &svd->vpage[page];
                        evpp = &svd->vpage[page + (len >> PAGESHIFT)];
                        for (; bvpp < evpp; bvpp++)
                                VPP_SETADVICE(bvpp, behav);
                        svd->advice = MADV_NORMAL;
                        break;
                case MADV_WILLNEED:     /* handled in memcntl */
                case MADV_DONTNEED:     /* handled in memcntl */
                case MADV_FREE:         /* handled above */
                        break;
                default:
                        err = EINVAL;
                }
        }
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        return (err);
}

/*
 * Create a vpage structure for this seg.
 */
static void
segvn_vpage(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vp, *evp;

        ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));

        /*
         * If no vpage structure exists, allocate one.  Copy the protections
         * and the advice from the segment itself to the individual pages.
         */
        if (svd->vpage == NULL) {
                svd->pageadvice = 1;
                svd->vpage = kmem_zalloc(seg_pages(seg) * sizeof (struct vpage),
                    KM_SLEEP);
                evp = &svd->vpage[seg_page(seg, seg->s_base + seg->s_size)];
```

```
                for (vp = svd->vpage; vp < evp; vp++) {
                        VPP_SETPROT(vp, svd->prot);
                        VPP_SETADVICE(vp, svd->advice);
                }
        }
}

/*
 * Dump the pages belonging to this segvn segment.
 */
static void
segvn_dump(struct seg *seg)
{
        struct segvn_data *svd;
        page_t *pp;
        struct anon_map *amp;
        ulong_t anon_index;
        struct vnode *vp;
        u_offset_t off, offset;
        pfn_t pfn;
        pgcnt_t page, npages;
        caddr_t addr;

        npages = seg_pages(seg);
        svd = (struct segvn_data *)seg->s_data;
        vp = svd->vp;
        off = offset = svd->offset;
        addr = seg->s_base;

        if ((amp = svd->amp) != NULL) {
                anon_index = svd->anon_index;
                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
        }

        for (page = 0; page < npages; page++, offset += PAGESIZE) {
                struct anon *ap;
                int we_own_it = 0;

                if (amp && (ap = anon_get_ptr(svd->amp->ahp, anon_index++))) {
                        swap_xlate_nopanic(ap, &vp, &off);
                } else {
                        vp = svd->vp;
                        off = offset;
                }

                /*
                 * If pp == NULL, the page either does not exist
                 * or is exclusively locked.  So determine if it
                 * exists before searching for it.
                 */

                if ((pp = page_lookup_nowait(vp, off, SE_SHARED)))
                        we_own_it = 1;
                else
                        pp = page_exists(vp, off);

                if (pp) {
                        pfn = page_pptonum(pp);
                        dump_addpage(seg->s_as, addr, pfn);
                        if (we_own_it)
                                page_unlock(pp);
                }
                addr += PAGESIZE;
                dump_timeleft = dump_timeout;
        }
        if (amp != NULL)
                ANON_LOCK_EXIT(&amp->a_rwlock);
}

#ifdef DEBUG
static uint32_t segvn_pglock_mtbf = 0;
#endif

#define PCACHE_SHWLIST          ((page_t *)-2)
#define NOPCACHE_SHWLIST        ((page_t *)-1)

/*
 * Lock/Unlock anon pages over a given range. Return shadow list. This routine
 * uses global segment pcache to cache shadow lists (i.e. pp arrays) of pages
 * to avoid the overhead of per page locking, unlocking for subsequent IOs to
 * the same parts of the segment. Currently shadow list creation is only
 * supported for pure anon segments. MAP_PRIVATE segment pcache entries are
 * tagged with segment pointer, starting virtual address and length. This
 * approach for MAP_SHARED segments may add many pcache entries for the same
 * set of pages and lead to long hash chains that decrease pcache lookup
 * performance. To avoid this issue for shared segments shared anon map and
 * starting anon index are used for pcache entry tagging. This allows all
 * segments to share pcache entries for the same anon range and reduces pcache
 * chain's length as well as memory overhead from duplicate shadow lists and
 * pcache entries.
 *
 * softlockcnt field in segvn_data structure counts the number of F_SOFTLOCK'd
 * pages via segvn_fault() and pagelock'd pages via this routine. But pagelock
 * part of softlockcnt accounting is done differently for private and shared
 * segments. In private segment case softlock is only incremented when a new
 * shadow list is created but not when an existing one is found via
 * seg_plookup(). pcache entries have reference count incremented/decremented
 * by each seg_plookup()/seg_pinactive() operation. Only entries that have 0
 * reference count can be purged (and purging is needed before segment can be
 * freed). When a private segment pcache entry is purged segvn_reclaim() will
 * decrement softlockcnt. Since in private segment case each of its pcache
 * entries only belongs to this segment we can expect that when
 * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
 * segment purge will succeed and softlockcnt will drop to 0. In shared
 * segment case reference count in pcache entry counts active locks from many
 * different segments so we can't expect segment purging to succeed even when
 * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
 * segment. To be able to determine when there're no pending pagelocks in
 * shared segment case we don't rely on purging to make softlockcnt drop to 0
 * but instead softlockcnt is incremented and decremented for every
 * segvn_pagelock(L_PAGELOCK/L_PAGEUNLOCK) call regardless if a new shadow
 * list was created or an existing one was found. When softlockcnt drops to 0
 * this segment no longer has any claims for pcached shadow lists and the
 * segment can be freed even if there're still active pcache entries
 * shared by this segment anon map. Shared segment pcache entries belong to
 * anon map and are typically removed when anon map is freed after all
 * processes destroy the segments that use this anon map.
 */
static int
segvn_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***ppp,
    enum lock_type type, enum seg_rw rw)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        size_t np;
        pgcnt_t adjustpages;
        pgcnt_t npages;
        ulong_t anon_index;
        uint_t protchk = (rw == S_READ) ? PROT_READ : PROT_WRITE;
        uint_t error;
        struct anon_map *amp;
        pgcnt_t anpgcnt;
```

```
        struct page **pplist, **pl, *pp;
        caddr_t a;
        size_t page;
        caddr_t lpgaddr, lpgeaddr;
        anon_sync_obj_t cookie;
        int anlock;
        struct anon_map *pamp;
        caddr_t paddr;
        seg_preclaim_cbfunc_t preclaim_callback;
        size_t pgsz;
        int use_pcache;
        size_t wlen;
        uint_t pflags = 0;
        int sftlck_sbase = 0;
        int sftlck_send = 0;

#ifdef DEBUG
        if (type == L_PAGELOCK && segvn_pglock_mtbf) {
                hrtime_t ts = gethrtime();
                if ((ts % segvn_pglock_mtbf) == 0) {
                        return (ENOTSUP);
                }
                if ((ts % segvn_pglock_mtbf) == 1) {
                        return (EFAULT);
                }
        }
#endif

        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
            "segvn_pagelock: start seg %p addr %p", seg, addr);

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

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

        /*
         * for now we only support pagelock to anon memory. We would have to
         * check protections for vnode objects and call into the vnode driver.
         * That's too much for a fast path. Let the fault entry point handle
         * it.
         */
        if (svd->vp != NULL) {
                if (type == L_PAGELOCK) {
                        error = ENOTSUP;
                        goto out;
                }
                panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
        }
        if ((amp = svd->amp) == NULL) {
                if (type == L_PAGELOCK) {
                        error = EFAULT;
                        goto out;
                }
                panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
        }
        if (rw != S_READ && rw != S_WRITE) {
                if (type == L_PAGELOCK) {
                        error = ENOTSUP;
                        goto out;
                }
                panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
        }

        if (seg->s_szc != 0) {
                /*
```

```
                 * We are adjusting the pagelock region to the large page size
                 * boundary because the unlocked part of a large page cannot
                 * be freed anyway unless all constituent pages of a large
                 * page are locked. Bigger regions reduce pcache chain length
                 * and improve lookup performance. The tradeoff is that the
                 * very first segvn_pagelock() call for a given page is more
                 * expensive if only 1 page_t is needed for IO. This is only
                 * an issue if pcache entry doesn't get reused by several
                 * subsequent calls. We optimize here for the case when pcache
                 * is heavily used by repeated IOs to the same address range.
                 *
                 * Note segment's page size cannot change while we are holding
                 * as lock.  And then it cannot change while softlockcnt is
                 * not 0. This will allow us to correctly recalculate large
                 * page size region for the matching pageunlock/reclaim call
                 * since as_pageunlock() caller must always match
                 * as_pagelock() call's addr and len.
                 *
                 * For pageunlock *ppp points to the pointer of page_t that
                 * corresponds to the real unadjusted start address. Similar
                 * for pagelock *ppp must point to the pointer of page_t that
                 * corresponds to the real unadjusted start address.
                 */
                pgsz = page_get_pagesize(seg->s_szc);
                CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
                adjustpages = btop((uintptr_t)(addr - lpgaddr));
        } else if (len < segvn_pglock_comb_thrshld) {
                lpgaddr = addr;
                lpgeaddr = addr + len;
                adjustpages = 0;
                pgsz = PAGESIZE;
        } else {
                /*
                 * Align the address range of large enough requests to allow
                 * combining of different shadow lists into 1 to reduce memory
                 * overhead from potentially overlapping large shadow lists
                 * (worst case is we have a 1MB IO into buffers with start
                 * addresses separated by 4K).  Alignment is only possible if
                 * padded chunks have sufficient access permissions. Note
                 * permissions won't change between L_PAGELOCK and
                 * L_PAGEUNLOCK calls since non 0 softlockcnt will force
                 * segvn_setprot() to wait until softlockcnt drops to 0. This
                 * allows us to determine in L_PAGEUNLOCK the same range we
                 * computed in L_PAGELOCK.
                 *
                 * If alignment is limited by segment ends set
                 * sftlck_sbase/sftlck_send flags. In L_PAGELOCK case when
                 * these flags are set bump softlockcnt_sbase/softlockcnt_send
                 * per segment counters. In L_PAGEUNLOCK case decrease
                 * softlockcnt_sbase/softlockcnt_send counters if
                 * sftlck_sbase/sftlck_send flags are set.  When
                 * softlockcnt_sbase/softlockcnt_send are non 0
                 * segvn_concat()/segvn_extend_prev()/segvn_extend_next()
                 * won't merge the segments. This restriction combined with
                 * restriction on segment unmapping and splitting for segments
                 * that have non 0 softlockcnt allows L_PAGEUNLOCK to
                 * correctly determine the same range that was previously
                 * locked by matching L_PAGELOCK.
                 */
                pflags = SEGP_PSHIFT | (segvn_pglock_comb_bshift << 16);
                pgsz = PAGESIZE;
                if (svd->type == MAP_PRIVATE) {
                        lpgaddr = (caddr_t)P2ALIGN((uintptr_t)addr,
                            segvn_pglock_comb_balign);
                        if (lpgaddr < seg->s_base) {
                                lpgaddr = seg->s_base;
```

```
                                sftlck_sbase = 1;
                        }
                } else {
                        ulong_t aix = svd->anon_index + seg_page(seg, addr);
                        ulong_t aaix = P2ALIGN(aix, segvn_pglock_comb_palign);
                        if (aaix < svd->anon_index) {
                                lpgaddr = seg->s_base;
                                sftlck_sbase = 1;
                        } else {
                                lpgaddr = addr - ptob(aix - aaix);
                                ASSERT(lpgaddr >= seg->s_base);
                        }
                }
                if (svd->pageprot && lpgaddr != addr) {
                        struct vpage *vp = &svd->vpage[seg_page(seg, lpgaddr)];
                        struct vpage *evp = &svd->vpage[seg_page(seg, addr)];
                        while (vp < evp) {
                                if ((VPP_PROT(vp) & protchk) == 0) {
                                        break;
                                }
                                vp++;
                        }
                        if (vp < evp) {
                                lpgaddr = addr;
                                pflags = 0;
                        }
                }
                lpgeaddr = addr + len;
                if (pflags) {
                        if (svd->type == MAP_PRIVATE) {
                                lpgeaddr = (caddr_t)P2ROUNDUP(
                                    (uintptr_t)lpgeaddr,
                                    segvn_pglock_comb_balign);
                        } else {
                                ulong_t aix = svd->anon_index +
                                    seg_page(seg, lpgeaddr);
                                ulong_t aaix = P2ROUNDUP(aix,
                                    segvn_pglock_comb_palign);
                                if (aaix < aix) {
                                        lpgeaddr = 0;
                                } else {
                                        lpgeaddr += ptob(aaix - aix);
                                }
                        }
                        if (lpgeaddr == 0 ||
                            lpgeaddr > seg->s_base + seg->s_size) {
                                lpgeaddr = seg->s_base + seg->s_size;
                                sftlck_send = 1;
                        }
                }
                if (svd->pageprot && lpgeaddr != addr + len) {
                        struct vpage *vp;
                        struct vpage *evp;

                        vp = &svd->vpage[seg_page(seg, addr + len)];
                        evp = &svd->vpage[seg_page(seg, lpgeaddr)];

                        while (vp < evp) {
                                if ((VPP_PROT(vp) & protchk) == 0) {
                                        break;
                                }
                                vp++;
                        }
                        if (vp < evp) {
                                lpgeaddr = addr + len;
                        }
                }
                adjustpages = btop((uintptr_t)(addr - lpgaddr));
        }

        /*
         * For MAP_SHARED segments we create pcache entries tagged by amp and
         * anon index so that we can share pcache entries with other segments
         * that map this amp.  For private segments pcache entries are tagged
         * with segment and virtual address.
         */
        if (svd->type == MAP_SHARED) {
                pamp = amp;
                paddr = (caddr_t)((lpgaddr - seg->s_base) +
                    ptob(svd->anon_index));
                preclaim_callback = shamp_reclaim;
        } else {
                pamp = NULL;
                paddr = lpgaddr;
                preclaim_callback = segvn_reclaim;
        }

        if (type == L_PAGEUNLOCK) {
                VM_STAT_ADD(segvnvmstats.pagelock[0]);

                /*
                 * update hat ref bits for /proc. We need to make sure
                 * that threads tracing the ref and mod bits of the
                 * address space get the right data.
                 * Note: page ref and mod bits are updated at reclaim time
                 */
                if (seg->s_as->a_vbits) {
                        for (a = addr; a < addr + len; a += PAGESIZE) {
                                if (rw == S_WRITE) {
                                        hat_setstat(seg->s_as, a,
                                            PAGESIZE, P_REF | P_MOD);
                                } else {
                                        hat_setstat(seg->s_as, a,
                                            PAGESIZE, P_REF);
                                }
                        }
                }

                /*
                 * Check the shadow list entry after the last page used in
                 * this IO request. If it’s NOPCACHE_SHWLIST the shadow list
                 * was not inserted into pcache and is not large page
                 * adjusted.  In this case call reclaim callback directly and
                 * don’t adjust the shadow list start and size for large
                 * pages.
                 */
                npages = btop(len);
                if ((*ppp)[npages] == NOPCACHE_SHWLIST) {
                        void *ptag;
                        if (pamp != NULL) {
                                ASSERT(svd->type == MAP_SHARED);
                                ptag = (void *)pamp;
                                paddr = (caddr_t)((addr - seg->s_base) +
                                    ptob(svd->anon_index));
                        } else {
                                ptag = (void *)seg;
                                paddr = addr;
                        }
                        (*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
                } else {
                        ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
                            IS_SWAPFSVP((*ppp)[npages]->p_vnode));
```

```
                        len = lpgeaddr - lpgaddr;
                        npages = btop(len);
                        seg_pinactive(seg, pamp, paddr, len,
                            *ppp - adjustpages, rw, pflags, preclaim_callback);
                }

                if (pamp != NULL) {
                        ASSERT(svd->type == MAP_SHARED);
                        ASSERT(svd->softlockcnt >= npages);
                        atomic_add_long((ulong_t *)&svd->softlockcnt, -npages);
                }

                if (sftlck_sbase) {
                        ASSERT(svd->softlockcnt_sbase > 0);
                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_sbase);
                }
                if (sftlck_send) {
                        ASSERT(svd->softlockcnt_send > 0);
                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_send);
                }

                /*
                 * If someone is blocked while unmapping, we purge
                 * segment page cache and thus reclaim pplist synchronously
                 * without waiting for seg_pasync_thread. This speeds up
                 * unmapping in cases where munmap(2) is called, while
                 * raw async i/o is still in progress or where a thread
                 * exits on data fault in a multithreaded application.
                 */
                if (AS_ISUNMAPWAIT(seg->s_as)) {
                        if (svd->softlockcnt == 0) {
                                mutex_enter(&seg->s_as->a_contents);
                                if (AS_ISUNMAPWAIT(seg->s_as)) {
                                        AS_CLRUNMAPWAIT(seg->s_as);
                                        cv_broadcast(&seg->s_as->a_cv);
                                }
                                mutex_exit(&seg->s_as->a_contents);
                        } else if (pamp == NULL) {
                                /*
                                 * softlockcnt is not 0 and this is a
                                 * MAP_PRIVATE segment. Try to purge its
                                 * pcache entries to reduce softlockcnt.
                                 * If it drops to 0 segvn_reclaim()
                                 * will wake up a thread waiting on
                                 * unmapwait flag.
                                 *
                                 * We don’t purge MAP_SHARED segments with non
                                 * 0 softlockcnt since IO is still in progress
                                 * for such segments.
                                 */
                                ASSERT(svd->type == MAP_PRIVATE);
                                segvn_purge(seg);
                        }
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_UNLOCK_END,
                    "segvn_pagelock: unlock seg %p addr %p", seg, addr);
                return (0);
        }

        /* The L_PAGELOCK case ... */

        VM_STAT_ADD(segvnvmstats.pagelock[1]);

        /*
         * For MAP_SHARED segments we have to check protections before
         * seg_plookup() since pcache entries may be shared by many segments
         * with potentially different page protections.
         */
        if (pamp != NULL) {
                ASSERT(svd->type == MAP_SHARED);
                if (svd->pageprot == 0) {
                        if ((svd->prot & protchk) == 0) {
                                error = EACCES;
                                goto out;
                        }
                } else {
                        /*
                         * check page protections
                         */
                        caddr_t ea;

                        if (seg->s_szc) {
                                a = lpgaddr;
                                ea = lpgeaddr;
                        } else {
                                a = addr;
                                ea = addr + len;
                        }
                        for (; a < ea; a += pgsz) {
                                struct vpage *vp;

                                ASSERT(seg->s_szc == 0 ||
                                    sameprot(seg, a, pgsz));
                                vp = &svd->vpage[seg_page(seg, a)];
                                if ((VPP_PROT(vp) & protchk) == 0) {
                                        error = EACCES;
                                        goto out;
                                }
                        }
                }
        }

        /*
         * try to find pages in segment page cache
         */
        pplist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
        if (pplist != NULL) {
                if (pamp != NULL) {
                        npages = btop((uintptr_t)(lpgeaddr - lpgaddr));
                        ASSERT(svd->type == MAP_SHARED);
                        atomic_add_long((ulong_t *)&svd->softlockcnt,
                            npages);
                }
                if (sftlck_sbase) {
                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
                }
                if (sftlck_send) {
                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                *ppp = pplist + adjustpages;
                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_HIT_END,
                    "segvn_pagelock: cache hit seg %p addr %p", seg, addr);
                return (0);
        }

        /*
         * For MAP_SHARED segments we already verified above that segment
         * protections allow this pagelock operation.
         */
        if (pamp == NULL) {
```

```
                ASSERT(svd->type == MAP_PRIVATE);
                if (svd->pageprot == 0) {
                        if ((svd->prot & protchk) == 0) {
                                error = EACCES;
                                goto out;
                        }
                        if (svd->prot & PROT_WRITE) {
                                wlen = lpgeaddr - lpgaddr;
                        } else {
                                wlen = 0;
                                ASSERT(rw == S_READ);
                        }
                } else {
                        int wcont = 1;
                        /*
                         * check page protections
                         */
                        for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
                                struct vpage *vp;

                                ASSERT(seg->s_szc == 0 ||
                                    sameprot(seg, a, pgsz));
                                vp = &svd->vpage[seg_page(seg, a)];
                                if ((VPP_PROT(vp) & protchk) == 0) {
                                        error = EACCES;
                                        goto out;
                                }
                                if (wcont && (VPP_PROT(vp) & PROT_WRITE)) {
                                        wlen += pgsz;
                                } else {
                                        wcont = 0;
                                        ASSERT(rw == S_READ);
                                }
                        }
                }
                ASSERT(rw == S_READ || wlen == lpgeaddr - lpgaddr);
                ASSERT(rw == S_WRITE || wlen <= lpgeaddr - lpgaddr);
        }

        /*
         * Only build large page adjusted shadow list if we expect to insert
         * it into pcache. For large enough pages it's a big overhead to
         * create a shadow list of the entire large page. But this overhead
         * should be amortized over repeated pcache hits on subsequent reuse
         * of this shadow list (IO into any range within this shadow list will
         * find it in pcache since we large page align the request for pcache
         * lookups). pcache performance is improved with bigger shadow lists
         * as it reduces the time to pcache the entire big segment and reduces
         * pcache chain length.
         */
        if (seg_pinsert_check(seg, pamp, paddr,
            lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
                addr = lpgaddr;
                len = lpgeaddr - lpgaddr;
                use_pcache = 1;
        } else {
                use_pcache = 0;
                /*
                 * Since this entry will not be inserted into the pcache, we
                 * will not do any adjustments to the starting address or
                 * size of the memory to be locked.
                 */
                adjustpages = 0;
        }
        npages = btop(len);
        pplist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
        pl = pplist;
        *ppp = pplist + adjustpages;
        /*
         * If use_pcache is 0 this shadow list is not large page adjusted.
         * Record this info in the last entry of shadow array so that
         * L_PAGEUNLOCK can determine if it should large page adjust the
         * address range to find the real range that was locked.
         */
        pl[npages] = use_pcache ? PCACHE_SHWLIST : NOPCACHE_SHWLIST;

        page = seg_page(seg, addr);
        anon_index = svd->anon_index + page;

        anlock = 0;
        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
        ASSERT(amp->a_szc >= seg->s_szc);
        anpgcnt = page_get_pagecnt(amp->a_szc);
        for (a = addr; a < addr + len; a += PAGESIZE, anon_index++) {
                struct anon *ap;
                struct vnode *vp;
                u_offset_t off;

                /*
                 * Lock and unlock anon array only once per large page.
                 * anon_array_enter() locks the root anon slot according to
                 * a_szc which can't change while anon map is locked.  We lock
                 * anon the first time through this loop and each time we
                 * reach anon index that corresponds to a root of a large
                 * page.
                 */
                if (a == addr || P2PHASE(anon_index, anpgcnt) == 0) {
                        ASSERT(anlock == 0);
                        anon_array_enter(amp, anon_index, &cookie);
                        anlock = 1;
                }
                ap = anon_get_ptr(amp->ahp, anon_index);

                /*
                 * We must never use seg_pcache for COW pages
                 * because we might end up with original page still
                 * lying in seg_pcache even after private page is
                 * created. This leads to data corruption as
                 * aio_write refers to the page still in cache
                 * while all other accesses refer to the private
                 * page.
                 */
                if (ap == NULL || ap->an_refcnt != 1) {
                        struct vpage *vpage;

                        if (seg->s_szc) {
                                error = EFAULT;
                                break;
                        }
                        if (svd->vpage != NULL) {
                                vpage = &svd->vpage[seg_page(seg, a)];
                        } else {
                                vpage = NULL;
                        }
                        ASSERT(anlock);
                        anon_array_exit(&cookie);
                        anlock = 0;
                        pp = NULL;
                        error = segvn_faultpage(seg->s_as->a_hat, seg, a, 0,
                            vpage, &pp, 0, F_INVAL, rw, 1);
                        if (error) {
```

```
                                error = fc_decode(error);
                                break;
                        }
                        anon_array_enter(amp, anon_index, &cookie);
                        anlock = 1;
                        ap = anon_get_ptr(amp->ahp, anon_index);
                        if (ap == NULL || ap->an_refcnt != 1) {
                                error = EFAULT;
                                break;
                        }
                }
                swap_xlate(ap, &vp, &off);
                pp = page_lookup_nowait(vp, off, SE_SHARED);
                if (pp == NULL) {
                        error = EFAULT;
                        break;
                }
                if (ap->an_pvp != NULL) {
                        anon_swap_free(ap, pp);
                }
                /*
                 * Unlock anon if this is the last slot in a large page.
                 */
                if (P2PHASE(anon_index, anpgcnt) == anpgcnt - 1) {
                        ASSERT(anlock);
                        anon_array_exit(&cookie);
                        anlock = 0;
                }
                *pplist++ = pp;
        }
        if (anlock) {           /* Ensure the lock is dropped */
                anon_array_exit(&cookie);
        }
        ANON_LOCK_EXIT(&amp->a_rwlock);

        if (a >= addr + len) {
                atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
                if (pamp != NULL) {
                        ASSERT(svd->type == MAP_SHARED);
                        atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
                            npages);
                        wlen = len;
                }
                if (sftlck_sbase) {
                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
                }
                if (sftlck_send) {
                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
                }
                if (use_pcache) {
                        (void) seg_pinsert(seg, pamp, paddr, len, wlen, pl,
                            rw, pflags, preclaim_callback);
                }
                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
                    "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
                return (0);
        }

        pplist = pl;
        np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
        while (np > (uint_t)0) {
                ASSERT(PAGE_LOCKED(*pplist));
                page_unlock(*pplist);
                np--;
                pplist++;
        }
        kmem_free(pl, sizeof (page_t *) * (npages + 1));
out:
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        *ppp = NULL;
        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
            "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
        return (error);
}

/*
 * purge any cached pages in the I/O page cache
 */
static void
segvn_purge(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;

        /*
         * pcache is only used by pure anon segments.
         */
        if (svd->amp == NULL || svd->vp != NULL) {
                return;
        }

        /*
         * For MAP_SHARED segments non 0 segment's softlockcnt means
         * active IO is still in progress via this segment. So we only
         * purge MAP_SHARED segments when their softlockcnt is 0.
         */
        if (svd->type == MAP_PRIVATE) {
                if (svd->softlockcnt) {
                        seg_ppurge(seg, NULL, 0);
                }
        } else if (svd->softlockcnt == 0 && svd->amp->a_softlockcnt != 0) {
                seg_ppurge(seg, svd->amp, 0);
        }
}

/*
 * If async argument is not 0 we are called from pcache async thread and don't
 * hold AS lock.
 */

/*ARGSUSED*/
static int
segvn_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
        enum seg_rw rw, int async)
{
        struct seg *seg = (struct seg *)ptag;
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        pgcnt_t np, npages;
        struct page **pl;

        npages = np = btop(len);
        ASSERT(npages);

        ASSERT(svd->vp == NULL && svd->amp != NULL);
        ASSERT(svd->softlockcnt >= npages);
        ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        pl = pplist;

        ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
        ASSERT(!async || pl[np] == PCACHE_SHWLIST);
```

```
        while (np > (uint_t)0) {
                if (rw == S_WRITE) {
                        hat_setrefmod(*pplist);
                } else {
                        hat_setref(*pplist);
                }
                page_unlock(*pplist);
                np--;
                pplist++;
        }

        kmem_free(pl, sizeof (page_t *) * (npages + 1));

        /*
         * If we are pcache async thread we don't hold AS lock. This means if
         * softlockcnt drops to 0 after the decrement below address space may
         * get freed. We can't allow it since after softlock derement to 0 we
         * still need to access as structure for possible wakeup of unmap
         * waiters. To prevent the disappearance of as we take this segment
         * segfree_syncmtx. segvn_free() also takes this mutex as a barrier to
         * make sure this routine completes before segment is freed.
         *
         * The second complication we have to deal with in async case is a
         * possibility of missed wake up of unmap wait thread. When we don't
         * hold as lock here we may take a_contents lock before unmap wait
         * thread that was first to see softlockcnt was still not 0. As a
         * result we'll fail to wake up an unmap wait thread. To avoid this
         * race we set nounmapwait flag in as structure if we drop softlockcnt
         * to 0 when we were called by pcache async thread.  unmapwait thread
         * will not block if this flag is set.
         */
        if (async) {
                mutex_enter(&svd->segfree_syncmtx);
        }

        if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -npages)) {
                if (async || AS_ISUNMAPWAIT(seg->s_as)) {
                        mutex_enter(&seg->s_as->a_contents);
                        if (async) {
                                AS_SETNOUNMAPWAIT(seg->s_as);
                        }
                        if (AS_ISUNMAPWAIT(seg->s_as)) {
                                AS_CLRUNMAPWAIT(seg->s_as);
                                cv_broadcast(&seg->s_as->a_cv);
                        }
                        mutex_exit(&seg->s_as->a_contents);
                }
        }

        if (async) {
                mutex_exit(&svd->segfree_syncmtx);
        }
        return (0);
}

/*ARGSUSED*/
static int
shamp_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
        enum seg_rw rw, int async)
{
        amp_t *amp = (amp_t *)ptag;
        pgcnt_t np, npages;
        struct page **pl;

        npages = np = btop(len);
        ASSERT(npages);
        ASSERT(amp->a_softlockcnt >= npages);

        pl = pplist;

        ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
        ASSERT(!async || pl[np] == PCACHE_SHWLIST);

        while (np > (uint_t)0) {
                if (rw == S_WRITE) {
                        hat_setrefmod(*pplist);
                } else {
                        hat_setref(*pplist);
                }
                page_unlock(*pplist);
                np--;
                pplist++;
        }

        kmem_free(pl, sizeof (page_t *) * (npages + 1));

        /*
         * If somebody sleeps in anonmap_purge() wake them up if a_softlockcnt
         * drops to 0. anon map can't be freed until a_softlockcnt drops to 0
         * and anonmap_purge() acquires a_purgemtx.
         */
        mutex_enter(&amp->a_purgemtx);
        if (!atomic_add_long_nv((ulong_t *)&amp->a_softlockcnt, -npages) &&
            amp->a_purgewait) {
                amp->a_purgewait = 0;
                cv_broadcast(&amp->a_purgecv);
        }
        mutex_exit(&amp->a_purgemtx);
        return (0);
}

/*
 * get a memory ID for an addr in a given segment
 *
 * XXX only creates PAGESIZE pages if anon slots are not initialized.
 * At fault time they will be relocated into larger pages.
 */
static int
segvn_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon     *ap = NULL;
        ulong_t         anon_index;
        struct anon_map *amp;
        anon_sync_obj_t cookie;

        if (svd->type == MAP_PRIVATE) {
                memidp->val[0] = (uintptr_t)seg->s_as;
                memidp->val[1] = (uintptr_t)addr;
                return (0);
        }

        if (svd->type == MAP_SHARED) {
                if (svd->vp) {
                        memidp->val[0] = (uintptr_t)svd->vp;
                        memidp->val[1] = (u_longlong_t)svd->offset +
                            (uintptr_t)(addr - seg->s_base);
                        return (0);
                } else {

                        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
                        if ((amp = svd->amp) != NULL) {
```

```
                                anon_index = svd->anon_index +
                                    seg_page(seg, addr);
                        }
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

                        ASSERT(amp != NULL);

                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        anon_array_enter(amp, anon_index, &cookie);
                        ap = anon_get_ptr(amp->ahp, anon_index);
                        if (ap == NULL) {
                                page_t          *pp;

                                pp = anon_zero(seg, addr, &ap, svd->cred);
                                if (pp == NULL) {
                                        anon_array_exit(&cookie);
                                        ANON_LOCK_EXIT(&amp->a_rwlock);
                                        return (ENOMEM);
                                }
                                ASSERT(anon_get_ptr(amp->ahp, anon_index)
                                    == NULL);
                                (void) anon_set_ptr(amp->ahp, anon_index,
                                    ap, ANON_SLEEP);
                                page_unlock(pp);
                        }

                        anon_array_exit(&cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);

                        memidp->val[0] = (uintptr_t)ap;
                        memidp->val[1] = (uintptr_t)addr & PAGEOFFSET;
                        return (0);
                }
        }
        return (EINVAL);
}

static int
sameprot(struct seg *seg, caddr_t a, size_t len)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct vpage *vpage;
        spgcnt_t pages = btop(len);
        uint_t prot;

        if (svd->pageprot == 0)
                return (1);

        ASSERT(svd->vpage != NULL);

        vpage = &svd->vpage[seg_page(seg, a)];
        prot = VPP_PROT(vpage);
        vpage++;
        pages--;
        while (pages-- > 0) {
                if (prot != VPP_PROT(vpage))
                        return (0);
                vpage++;
        }
        return (1);
}

/*
 * Get memory allocation policy info for specified address in given segment
 */
static lgrp_mem_policy_info_t *
```

```
segvn_getpolicy(struct seg *seg, caddr_t addr)
{
        struct anon_map         *amp;
        ulong_t                 anon_index;
        lgrp_mem_policy_info_t  *policy_info;
        struct segvn_data       *svn_data;
        u_offset_t              vn_off;
        vnode_t                 *vp;

        ASSERT(seg != NULL);

        svn_data = (struct segvn_data *)seg->s_data;
        if (svn_data == NULL)
                return (NULL);

        /*
         * Get policy info for private or shared memory
         */
        if (svn_data->type != MAP_SHARED) {
                if (svn_data->tr_state != SEGVN_TR_ON) {
                        policy_info = &svn_data->policy_info;
                } else {
                        policy_info = &svn_data->tr_policy_info;
                        ASSERT(policy_info->mem_policy ==
                            LGRP_MEM_POLICY_NEXT_SEG);
                }
        } else {
                amp = svn_data->amp;
                anon_index = svn_data->anon_index + seg_page(seg, addr);
                vp = svn_data->vp;
                vn_off = svn_data->offset + (uintptr_t)(addr - seg->s_base);
                policy_info = lgrp_shm_policy_get(amp, anon_index, vp, vn_off);
        }

        return (policy_info);
}

/*ARGSUSED*/
static int
segvn_capable(struct seg *seg, segcapability_t capability)
{
        return (0);
}

/*
 * Bind text vnode segment to an amp. If we bind successfully mappings will be
 * established to per vnode mapping per lgroup amp pages instead of to vnode
 * pages. There's one amp per vnode text mapping per lgroup. Many processes
 * may share the same text replication amp. If a suitable amp doesn't already
 * exist in svntr hash table create a new one.  We may fail to bind to amp if
 * segment is not eligible for text replication.  Code below first checks for
 * these conditions. If binding is successful segment tr_state is set to on
 * and svd->amp points to the amp to use. Otherwise tr_state is set to off and
 * svd->amp remains as NULL.
 */
static void
segvn_textrepl(struct seg *seg)
{
        struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
        vnode_t                 *vp = svd->vp;
        u_offset_t              off = svd->offset;
        size_t                  size = seg->s_size;
        u_offset_t              eoff = off + size;
        uint_t                  szc = seg->s_szc;
        ulong_t                 hash = SVNTR_HASH_FUNC(vp);
        svntr_t                 *svntrp;
```

```
        struct vattr            va;
        proc_t                  *p = seg->s_as->a_proc;
        lgrp_id_t               lgrp_id;
        lgrp_id_t               olid;
        int                     first;
        struct anon_map         *amp;

        ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
        ASSERT(p != NULL);
        ASSERT(svd->tr_state == SEGVN_TR_INIT);
        ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
        ASSERT(svd->flags & MAP_TEXT);
        ASSERT(svd->type == MAP_PRIVATE);
        ASSERT(vp != NULL && svd->amp == NULL);
        ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
        ASSERT(!(svd->flags & MAP_NORESERVE) && svd->swresv == 0);
        ASSERT(seg->s_as != &kas);
        ASSERT(off < eoff);
        ASSERT(svntr_hashtab != NULL);

        /*
         * If numa optimizations are no longer desired bail out.
         */
        if (!lgrp_optimizations()) {
                svd->tr_state = SEGVN_TR_OFF;
                return;
        }

        /*
         * Avoid creating anon maps with size bigger than the file size.
         * If VOP_GETATTR() call fails bail out.
         */
        va.va_mask = AT_SIZE | AT_MTIME | AT_CTIME;
        if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL) != 0) {
                svd->tr_state = SEGVN_TR_OFF;
                SEGVN_TR_ADDSTAT(gaerr);
                return;
        }
        if (btopr(va.va_size) < btopr(eoff)) {
                svd->tr_state = SEGVN_TR_OFF;
                SEGVN_TR_ADDSTAT(overmap);
                return;
        }

        /*
         * VVMEXEC may not be set yet if exec() prefaults text segment. Set
         * this flag now before vn_is_mapped(V_WRITE) so that MAP_SHARED
         * mapping that checks if trcache for this vnode needs to be
         * invalidated can't miss us.
         */
        if (!(vp->v_flag & VVMEXEC)) {
                mutex_enter(&vp->v_lock);
                vp->v_flag |= VVMEXEC;
                mutex_exit(&vp->v_lock);
        }
        mutex_enter(&svntr_hashtab[hash].tr_lock);
        /*
         * Bail out if potentially MAP_SHARED writable mappings exist to this
         * vnode.  We don't want to use old file contents from existing
         * replicas if this mapping was established after the original file
         * was changed.
         */
        if (vn_is_mapped(vp, V_WRITE)) {
                mutex_exit(&svntr_hashtab[hash].tr_lock);
                svd->tr_state = SEGVN_TR_OFF;
                SEGVN_TR_ADDSTAT(wrcnt);
                return;
        }
        svntrp = svntr_hashtab[hash].tr_head;
        for (; svntrp != NULL; svntrp = svntrp->tr_next) {
                ASSERT(svntrp->tr_refcnt != 0);
                if (svntrp->tr_vp != vp) {
                        continue;
                }

                /*
                 * Bail out if the file or its attributes were changed after
                 * this replication entry was created since we need to use the
                 * latest file contents. Note that mtime test alone is not
                 * sufficient because a user can explicitly change mtime via
                 * utimes(2) interfaces back to the old value after modifiying
                 * the file contents. To detect this case we also have to test
                 * ctime which among other things records the time of the last
                 * mtime change by utimes(2). ctime is not changed when the file
                 * is only read or executed so we expect that typically existing
                 * replication amp’s can be used most of the time.
                 */
                if (!svntrp->tr_valid ||
                    svntrp->tr_mtime.tv_sec != va.va_mtime.tv_sec ||
                    svntrp->tr_mtime.tv_nsec != va.va_mtime.tv_nsec ||
                    svntrp->tr_ctime.tv_sec != va.va_ctime.tv_sec ||
                    svntrp->tr_ctime.tv_nsec != va.va_ctime.tv_nsec) {
                        mutex_exit(&svntr_hashtab[hash].tr_lock);
                        svd->tr_state = SEGVN_TR_OFF;
                        SEGVN_TR_ADDSTAT(stale);
                        return;
                }
                /*
                 * if off, eoff and szc match current segment we found the
                 * existing entry we can use.
                 */
                if (svntrp->tr_off == off && svntrp->tr_eoff == eoff &&
                    svntrp->tr_szc == szc) {
                        break;
                }
                /*
                 * Don’t create different but overlapping in file offsets
                 * entries to avoid replication of the same file pages more
                 * than once per lgroup.
                 */
                if ((off >= svntrp->tr_off && off < svntrp->tr_eoff) ||
                    (eoff > svntrp->tr_off && eoff <= svntrp->tr_eoff)) {
                        mutex_exit(&svntr_hashtab[hash].tr_lock);
                        svd->tr_state = SEGVN_TR_OFF;
                        SEGVN_TR_ADDSTAT(overlap);
                        return;
                }
        }
        /*
         * If we didn’t find existing entry create a new one.
         */
        if (svntrp == NULL) {
                svntrp = kmem_cache_alloc(svntr_cache, KM_NOSLEEP);
                if (svntrp == NULL) {
                        mutex_exit(&svntr_hashtab[hash].tr_lock);
                        svd->tr_state = SEGVN_TR_OFF;
                        SEGVN_TR_ADDSTAT(nokmem);
                        return;
                }
#ifdef DEBUG
                {
```

```
                        lgrp_id_t i;
                        for (i = 0; i < NLGRPS_MAX; i++) {
                                ASSERT(svntrp->tr_amp[i] == NULL);
                        }
                }
#endif /* DEBUG */
                svntrp->tr_vp = vp;
                svntrp->tr_off = off;
                svntrp->tr_eoff = eoff;
                svntrp->tr_szc = szc;
                svntrp->tr_valid = 1;
                svntrp->tr_mtime = va.va_mtime;
                svntrp->tr_ctime = va.va_ctime;
                svntrp->tr_refcnt = 0;
                svntrp->tr_next = svntr_hashtab[hash].tr_head;
                svntr_hashtab[hash].tr_head = svntrp;
        }
        first = 1;
again:
        /*
         * We want to pick a replica with pages on main thread’s (t_tid = 1,
         * aka T1) lgrp. Currently text replication is only optimized for
         * workloads that either have all threads of a process on the same
         * lgrp or execute their large text primarily on main thread.
         */
        lgrp_id = p->p_t1_lgrpid;
        if (lgrp_id == LGRP_NONE) {
                /*
                 * In case exec() prefaults text on non main thread use
                 * current thread lgrpid.  It will become main thread anyway
                 * soon.
                 */
                lgrp_id = lgrp_home_id(curthread);
        }
        /*
         * Set p_tr_lgrpid to lgrpid if it hasn’t been set yet.  Otherwise
         * just set it to NLGRPS_MAX if it’s different from current process T1
         * home lgrp.  p_tr_lgrpid is used to detect if process uses text
         * replication and T1 new home is different from lgrp used for text
         * replication. When this happens asyncronous segvn thread rechecks if
         * segments should change lgrps used for text replication.  If we fail
         * to set p_tr_lgrpid with atomic_cas_32 then set it to NLGRPS_MAX
         * without cas if it’s not already NLGRPS_MAX and not equal lgrp_id
         * we want to use.  We don’t need to use cas in this case because
         * another thread that races in between our non atomic check and set
         * may only change p_tr_lgrpid to NLGRPS_MAX at this point.
         */
        ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
        olid = p->p_tr_lgrpid;
        if (lgrp_id != olid && olid != NLGRPS_MAX) {
                lgrp_id_t nlid = (olid == LGRP_NONE) ? lgrp_id : NLGRPS_MAX;
                if (atomic_cas_32((uint32_t *)&p->p_tr_lgrpid, olid, nlid) !=
                    olid) {
                        olid = p->p_tr_lgrpid;
                        ASSERT(olid != LGRP_NONE);
                        if (olid != lgrp_id && olid != NLGRPS_MAX) {
                                p->p_tr_lgrpid = NLGRPS_MAX;
                        }
                }
                ASSERT(p->p_tr_lgrpid != LGRP_NONE);
                membar_producer();
                /*
                 * lgrp_move_thread() won’t schedule async recheck after
                 * p->p_t1_lgrpid update unless p->p_tr_lgrpid is not
                 * LGRP_NONE. Recheck p_t1_lgrpid once now that p->p_tr_lgrpid
                 * is not LGRP_NONE.
                 */
                if (first && p->p_t1_lgrpid != LGRP_NONE &&
                    p->p_t1_lgrpid != lgrp_id) {
                        first = 0;
                        goto again;
                }
        }
        /*
         * If no amp was created yet for lgrp_id create a new one as long as
         * we have enough memory to afford it.
         */
        if ((amp = svntrp->tr_amp[lgrp_id]) == NULL) {
                size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
                if (trmem > segvn_textrepl_max_bytes) {
                        SEGVN_TR_ADDSTAT(normem);
                        goto fail;
                }
                if (anon_try_resv_zone(size, NULL) == 0) {
                        SEGVN_TR_ADDSTAT(noanon);
                        goto fail;
                }
                amp = anonmap_alloc(size, size, ANON_NOSLEEP);
                if (amp == NULL) {
                        anon_unresv_zone(size, NULL);
                        SEGVN_TR_ADDSTAT(nokmem);
                        goto fail;
                }
                ASSERT(amp->refcnt == 1);
                amp->a_szc = szc;
                svntrp->tr_amp[lgrp_id] = amp;
                SEGVN_TR_ADDSTAT(newamp);
        }
        svntrp->tr_refcnt++;
        ASSERT(svd->svn_trnext == NULL);
        ASSERT(svd->svn_trprev == NULL);
        svd->svn_trnext = svntrp->tr_svnhead;
        svd->svn_trprev = NULL;
        if (svntrp->tr_svnhead != NULL) {
                svntrp->tr_svnhead->svn_trprev = svd;
        }
        svntrp->tr_svnhead = svd;
        ASSERT(amp->a_szc == szc && amp->size == size && amp->swresv == size);
        ASSERT(amp->refcnt >= 1);
        svd->amp = amp;
        svd->anon_index = 0;
        svd->tr_policy_info.mem_policy = LGRP_MEM_POLICY_NEXT_SEG;
        svd->tr_policy_info.mem_lgrpid = lgrp_id;
        svd->tr_state = SEGVN_TR_ON;
        mutex_exit(&svntr_hashtab[hash].tr_lock);
        SEGVN_TR_ADDSTAT(repl);
        return;
fail:
        ASSERT(segvn_textrepl_bytes >= size);
        atomic_add_long(&segvn_textrepl_bytes, -size);
        ASSERT(svntrp != NULL);
        ASSERT(svntrp->tr_amp[lgrp_id] == NULL);
        if (svntrp->tr_refcnt == 0) {
                ASSERT(svntrp == svntr_hashtab[hash].tr_head);
                svntr_hashtab[hash].tr_head = svntrp->tr_next;
                mutex_exit(&svntr_hashtab[hash].tr_lock);
                kmem_cache_free(svntr_cache, svntrp);
        } else {
                mutex_exit(&svntr_hashtab[hash].tr_lock);
        }
        svd->tr_state = SEGVN_TR_OFF;
}
```

```
/*
 * Convert seg back to regular vnode mapping seg by unbinding it from its text
 * replication amp.  This routine is most typically called when segment is
 * unmapped but can also be called when segment no longer qualifies for text
 * replication (e.g. due to protection changes). If unload_unmap is set use
 * HAT_UNLOAD_UNMAP flag in hat_unload_callback().  If we are the last user of
 * svntr free all its anon maps and remove it from the hash table.
 */
static void
segvn_textunrepl(struct seg *seg, int unload_unmap)
{
        struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
        vnode_t                 *vp = svd->vp;
        u_offset_t              off = svd->offset;
        size_t                  size = seg->s_size;
        u_offset_t              eoff = off + size;
        uint_t                  szc = seg->s_szc;
        ulong_t                 hash = SVNTR_HASH_FUNC(vp);
        svntr_t                 *svntrp;
        svntr_t                 **prv_svntrp;
        lgrp_id_t               lgrp_id = svd->tr_policy_info.mem_lgrpid;
        lgrp_id_t               i;

        ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
            SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
        ASSERT(svd->tr_state == SEGVN_TR_ON);
        ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
        ASSERT(svd->amp != NULL);
        ASSERT(svd->amp->refcnt >= 1);
        ASSERT(svd->anon_index == 0);
        ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
        ASSERT(svntr_hashtab != NULL);

        mutex_enter(&svntr_hashtab[hash].tr_lock);
        prv_svntrp = &svntr_hashtab[hash].tr_head;
        for (; (svntrp = *prv_svntrp) != NULL; prv_svntrp = &svntrp->tr_next) {
                ASSERT(svntrp->tr_refcnt != 0);
                if (svntrp->tr_vp == vp && svntrp->tr_off == off &&
                    svntrp->tr_eoff == eoff && svntrp->tr_szc == szc) {
                        break;
                }
        }
        if (svntrp == NULL) {
                panic("segvn_textunrepl: svntr record not found");
        }
        if (svntrp->tr_amp[lgrp_id] != svd->amp) {
                panic("segvn_textunrepl: amp mismatch");
        }
        svd->tr_state = SEGVN_TR_OFF;
        svd->amp = NULL;
        if (svd->svn_trprev == NULL) {
                ASSERT(svntrp->tr_svnhead == svd);
                svntrp->tr_svnhead = svd->svn_trnext;
                if (svntrp->tr_svnhead != NULL) {
                        svntrp->tr_svnhead->svn_trprev = NULL;
                }
                svd->svn_trnext = NULL;
        } else {
                svd->svn_trprev->svn_trnext = svd->svn_trnext;
                if (svd->svn_trnext != NULL) {
                        svd->svn_trnext->svn_trprev = svd->svn_trprev;
                        svd->svn_trnext = NULL;
                }
                svd->svn_trprev = NULL;
        }
        if (--svntrp->tr_refcnt) {
                mutex_exit(&svntr_hashtab[hash].tr_lock);
                goto done;
        }
        *prv_svntrp = svntrp->tr_next;
        mutex_exit(&svntr_hashtab[hash].tr_lock);
        for (i = 0; i < NLGRPS_MAX; i++) {
                struct anon_map *amp = svntrp->tr_amp[i];
                if (amp == NULL) {
                        continue;
                }
                ASSERT(amp->refcnt == 1);
                ASSERT(amp->swresv == size);
                ASSERT(amp->size == size);
                ASSERT(amp->a_szc == szc);
                if (amp->a_szc != 0) {
                        anon_free_pages(amp->ahp, 0, size, szc);
                } else {
                        anon_free(amp->ahp, 0, size);
                }
                svntrp->tr_amp[i] = NULL;
                ASSERT(segvn_textrepl_bytes >= size);
                atomic_add_long(&segvn_textrepl_bytes, -size);
                anon_unresv_zone(amp->swresv, NULL);
                amp->refcnt = 0;
                anonmap_free(amp);
        }
        kmem_cache_free(svntr_cache, svntrp);
done:
        hat_unload_callback(seg->s_as->a_hat, seg->s_base, size,
            unload_unmap ? HAT_UNLOAD_UNMAP : 0, NULL);
}

/*
 * This is called when a MAP_SHARED writable mapping is created to a vnode
 * that is currently used for execution (VVMEXEC flag is set). In this case we
 * need to prevent further use of existing replicas.
 */
static void
segvn_inval_trcache(vnode_t *vp)
{
        ulong_t                 hash = SVNTR_HASH_FUNC(vp);
        svntr_t                 *svntrp;

        ASSERT(vp->v_flag & VVMEXEC);

        if (svntr_hashtab == NULL) {
                return;
        }

        mutex_enter(&svntr_hashtab[hash].tr_lock);
        svntrp = svntr_hashtab[hash].tr_head;
        for (; svntrp != NULL; svntrp = svntrp->tr_next) {
                ASSERT(svntrp->tr_refcnt != 0);
                if (svntrp->tr_vp == vp && svntrp->tr_valid) {
                        svntrp->tr_valid = 0;
                }
        }
        mutex_exit(&svntr_hashtab[hash].tr_lock);
}

static void
segvn_trasync_thread(void)
{
        callb_cpr_t cpr_info;
```

```
        kmutex_t cpr_lock;      /* just for CPR stuff */

        mutex_init(&cpr_lock, NULL, MUTEX_DEFAULT, NULL);

        CALLB_CPR_INIT(&cpr_info, &cpr_lock,
            callb_generic_cpr, "segvn_async");

        if (segvn_update_textrepl_interval == 0) {
                segvn_update_textrepl_interval = segvn_update_tr_time * hz;
        } else {
                segvn_update_textrepl_interval *= hz;
        }
        (void) timeout(segvn_trupdate_wakeup, NULL,
            segvn_update_textrepl_interval);

        for (;;) {
                mutex_enter(&cpr_lock);
                CALLB_CPR_SAFE_BEGIN(&cpr_info);
                mutex_exit(&cpr_lock);
                sema_p(&segvn_trasync_sem);
                mutex_enter(&cpr_lock);
                CALLB_CPR_SAFE_END(&cpr_info, &cpr_lock);
                mutex_exit(&cpr_lock);
                segvn_trupdate();
        }
}

static uint64_t segvn_lgrp_trthr_migrs_snpsht = 0;

static void
segvn_trupdate_wakeup(void *dummy)
{
        uint64_t cur_lgrp_trthr_migrs = lgrp_get_trthr_migrations();

        if (cur_lgrp_trthr_migrs != segvn_lgrp_trthr_migrs_snpsht) {
                segvn_lgrp_trthr_migrs_snpsht = cur_lgrp_trthr_migrs;
                sema_v(&segvn_trasync_sem);
        }

        if (!segvn_disable_textrepl_update &&
            segvn_update_textrepl_interval != 0) {
                (void) timeout(segvn_trupdate_wakeup, dummy,
                    segvn_update_textrepl_interval);
        }
}

static void
segvn_trupdate(void)
{
        ulong_t         hash;
        svntr_t         *svntrp;
        segvn_data_t    *svd;

        ASSERT(svntr_hashtab != NULL);

        for (hash = 0; hash < svntr_hashtab_sz; hash++) {
                mutex_enter(&svntr_hashtab[hash].tr_lock);
                svntrp = svntr_hashtab[hash].tr_head;
                for (; svntrp != NULL; svntrp = svntrp->tr_next) {
                        ASSERT(svntrp->tr_refcnt != 0);
                        svd = svntrp->tr_svnhead;
                        for (; svd != NULL; svd = svd->svn_trnext) {
                                segvn_trupdate_seg(svd->seg, svd, svntrp,
                                    hash);
                        }
                }
                mutex_exit(&svntr_hashtab[hash].tr_lock);
        }
}

static void
segvn_trupdate_seg(struct seg *seg,
        segvn_data_t *svd,
        svntr_t *svntrp,
        ulong_t hash)
{
        proc_t                  *p;
        lgrp_id_t               lgrp_id;
        struct as               *as;
        size_t                  size;
        struct anon_map         *amp;

        ASSERT(svd->vp != NULL);
        ASSERT(svd->vp == svntrp->tr_vp);
        ASSERT(svd->offset == svntrp->tr_off);
        ASSERT(svd->offset + seg->s_size == svntrp->tr_eoff);
        ASSERT(seg != NULL);
        ASSERT(svd->seg == seg);
        ASSERT(seg->s_data == (void *)svd);
        ASSERT(seg->s_szc == svntrp->tr_szc);
        ASSERT(svd->tr_state == SEGVN_TR_ON);
        ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
        ASSERT(svd->amp != NULL);
        ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
        ASSERT(svd->tr_policy_info.mem_lgrpid != LGRP_NONE);
        ASSERT(svd->tr_policy_info.mem_lgrpid < NLGRPS_MAX);
        ASSERT(svntrp->tr_amp[svd->tr_policy_info.mem_lgrpid] == svd->amp);
        ASSERT(svntrp->tr_refcnt != 0);
        ASSERT(mutex_owned(&svntr_hashtab[hash].tr_lock));

        as = seg->s_as;
        ASSERT(as != NULL && as != &kas);
        p = as->a_proc;
        ASSERT(p != NULL);
        ASSERT(p->p_tr_lgrpid != LGRP_NONE);
        lgrp_id = p->p_t1_lgrpid;
        if (lgrp_id == LGRP_NONE) {
                return;
        }
        ASSERT(lgrp_id < NLGRPS_MAX);
        if (svd->tr_policy_info.mem_lgrpid == lgrp_id) {
                return;
        }

        /*
         * Use tryenter locking since we are locking as/seg and svntr hash
         * lock in reverse from syncrounous thread order.
         */
        if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_READER)) {
                SEGVN_TR_ADDSTAT(nolock);
                if (segvn_lgrp_trthr_migrs_snpsht) {
                        segvn_lgrp_trthr_migrs_snpsht = 0;
                }
                return;
        }
        if (!SEGVN_LOCK_TRYENTER(seg->s_as, &svd->lock, RW_WRITER)) {
                AS_LOCK_EXIT(as, &as->a_lock);
                SEGVN_TR_ADDSTAT(nolock);
                if (segvn_lgrp_trthr_migrs_snpsht) {
                        segvn_lgrp_trthr_migrs_snpsht = 0;
                }
                return;
```

```
        }
        size = seg->s_size;
        if (svntrp->tr_amp[lgrp_id] == NULL) {
                size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
                if (trmem > segvn_textrepl_max_bytes) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        AS_LOCK_EXIT(as, &as->a_lock);
                        atomic_add_long(&segvn_textrepl_bytes, -size);
                        SEGVN_TR_ADDSTAT(normem);
                        return;
                }
                if (anon_try_resv_zone(size, NULL) == 0) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        AS_LOCK_EXIT(as, &as->a_lock);
                        atomic_add_long(&segvn_textrepl_bytes, -size);
                        SEGVN_TR_ADDSTAT(noanon);
                        return;
                }
                amp = anonmap_alloc(size, size, KM_NOSLEEP);
                if (amp == NULL) {
                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
                        AS_LOCK_EXIT(as, &as->a_lock);
                        atomic_add_long(&segvn_textrepl_bytes, -size);
                        anon_unresv_zone(size, NULL);
                        SEGVN_TR_ADDSTAT(nokmem);
                        return;
                }
                ASSERT(amp->refcnt == 1);
                amp->a_szc = seg->s_szc;
                svntrp->tr_amp[lgrp_id] = amp;
        }
        /*
         * We don't need to drop the bucket lock but here we give other
         * threads a chance.  svntr and svd can't be unlinked as long as
         * segment lock is held as a writer and AS held as well.  After we
         * retake bucket lock we'll continue from where we left. We'll be able
         * to reach the end of either list since new entries are always added
         * to the beginning of the lists.
         */
        mutex_exit(&svntr_hashtab[hash].tr_lock);
        hat_unload_callback(as->a_hat, seg->s_base, size, 0, NULL);
        mutex_enter(&svntr_hashtab[hash].tr_lock);

        ASSERT(svd->tr_state == SEGVN_TR_ON);
        ASSERT(svd->amp != NULL);
        ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
        ASSERT(svd->tr_policy_info.mem_lgrpid != lgrp_id);
        ASSERT(svd->amp != svntrp->tr_amp[lgrp_id]);

        svd->tr_policy_info.mem_lgrpid = lgrp_id;
        svd->amp = svntrp->tr_amp[lgrp_id];
        p->p_tr_lgrpid = NLGRPS_MAX;
        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
        AS_LOCK_EXIT(as, &as->a_lock);

        ASSERT(svntrp->tr_refcnt != 0);
        ASSERT(svd->vp == svntrp->tr_vp);
        ASSERT(svd->tr_policy_info.mem_lgrpid == lgrp_id);
        ASSERT(svd->amp != NULL && svd->amp == svntrp->tr_amp[lgrp_id]);
        ASSERT(svd->seg == seg);
        ASSERT(svd->tr_state == SEGVN_TR_ON);

        SEGVN_TR_ADDSTAT(asyncrepl);
}
```