```
**********************************************************
    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,
```

```
127                     struct vnode **vpp);
128 static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,
129                     uint_t behav);
130 static void     segvn_dump(struct seg *seg);
131 static int      segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
132                     struct page ***ppp, enum lock_type type, enum seg_rw rw);
133 static int      segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
134                     uint_t szc);
135 static int      segvn_getmemid(struct seg *seg, caddr_t addr,
136                     memid_t *memidp);
137 static lgrp_mem_policy_info_t   *segvn_getpolicy(struct seg *, caddr_t);
138 static int      segvn_capable(struct seg *seg, segcapability_t capable);

140 struct  seg_ops segvn_ops = {
141         segvn_dup,
142         segvn_unmap,
143         segvn_free,
144         segvn_fault,
145         segvn_faulta,
146         segvn_setprot,
147         segvn_checkprot,
148         segvn_kluster,
149         segvn_swapout,
150         segvn_sync,
151         segvn_incore,
152         segvn_lockop,
153         segvn_getprot,
154         segvn_getoffset,
155         segvn_gettype,
156         segvn_getvp,
157         segvn_advise,
158         segvn_dump,
159         segvn_pagelock,
160         segvn_setpagesize,
161         segvn_getmemid,
162         segvn_getpolicy,
163         segvn_capable,
164 };

166 /*
167  * Common zfod structures, provided as a shorthand for others to use.
168  */
169 static segvn_crargs_t zfod_segvn_crargs =
170         SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
171 static segvn_crargs_t kzfod_segvn_crargs =
172         SEGVN_ZFOD_ARGS(PROT_ZFOD & ~PROT_USER,
173         PROT_ALL & ~PROT_USER);
174 static segvn_crargs_t stack_noexec_crargs =
175         SEGVN_ZFOD_ARGS(PROT_ZFOD & ~PROT_EXEC, PROT_ALL);

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

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

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

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

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



```
193                     struct segvn_crargs *, size_t);
194 static int      segvn_extend_next(struct seg *, struct seg *,
195                     struct segvn_crargs *, size_t);
196 static void     segvn_softunlock(struct seg *, caddr_t, size_t, enum seg_rw);
197 static void     segvn_pagelist_rele(page_t **);
198 static void     segvn_setvnode_mpss(vnode_t *);
199 static void     segvn_relocate_pages(page_t **, page_t *);
200 static int      segvn_full_szcpages(page_t **, uint_t, int *, uint_t *);
201 static int      segvn_fill_vp_pages(struct segvn_data *, vnode_t *, u_offset_t,
202     uint_t, page_t **, page_t **, uint_t *, int *);
203 static faultcode_t segvn_fault_vnodepages(struct hat *, struct seg *, caddr_t,
204     caddr_t, enum fault_type, enum seg_rw, caddr_t, caddr_t, int);
205 static faultcode_t segvn_fault_anonpages(struct hat *, struct seg *, caddr_t,
206     caddr_t, enum fault_type, enum seg_rw, caddr_t, caddr_t, int);
207 static faultcode_t segvn_faultpage(struct hat *, struct seg *, caddr_t,
208     u_offset_t, struct vpage *, page_t **, uint_t,
209     enum fault_type, enum seg_rw, int);
210 static void     segvn_vpage(struct seg *);
211 static size_t   segvn_count_swap_by_vpages(struct seg *);

213 static void segvn_purge(struct seg *seg);
214 static int segvn_reclaim(void *, caddr_t, size_t, struct page **,
215     enum seg_rw, int);
216 static int shamp_reclaim(void *, caddr_t, size_t, struct page **,
217     enum seg_rw, int);

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

221 static int segvn_demote_range(struct seg *, caddr_t, size_t, int, uint_t);
222 static int segvn_clrszc(struct seg *);
223 static struct seg *segvn_split_seg(struct seg *, caddr_t);
224 static int segvn_claim_pages(struct seg *, struct vpage *, u_offset_t,
225     ulong_t, uint_t);

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

230 static struct kmem_cache *segvn_cache;
231 static struct kmem_cache **segvn_szc_cache;

233 #ifdef VM_STATS
234 static struct segvnvmstats_str {
235         ulong_t fill_vp_pages[31];
236         ulong_t fltvnpages[49];
237         ulong_t fullszcpages[10];
238         ulong_t relocatepages[3];
239         ulong_t fltanpages[17];
240         ulong_t pagelock[2];
241         ulong_t demoterange[3];
242 } segvnvmstats;
243 #endif /* VM_STATS */

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

248 #define CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr) {          \
249                 if ((len) != 0) {                                           \
250                         lpgaddr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);  \
251                         ASSERT(lpgaddr >= (seg)->s_base);                   \
252                         lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)((addr) +    \
253                             (len)), pgsz);                                  \
254                         ASSERT(lpgeaddr > lpgaddr);                         \
255                         ASSERT(lpgeaddr <= (seg)->s_base + (seg)->s_size);    \
256                 } else {                                                    \
257                         lpgeaddr = lpgaddr = (addr);                        \
258                 }                                                           \
```

```
259         }

261 /*ARGSUSED*/
262 static int
263 segvn_cache_constructor(void *buf, void *cdrarg, int kmflags)
264 {
265         struct segvn_data *svd = buf;

267         rw_init(&svd->lock, NULL, RW_DEFAULT, NULL);
268         mutex_init(&svd->segfree_syncmtx, NULL, MUTEX_DEFAULT, NULL);
269         svd->svn_trnext = svd->svn_trprev = NULL;
270         return (0);
271 }

273 /*ARGSUSED1*/
274 static void
275 segvn_cache_destructor(void *buf, void *cdrarg)
276 {
277         struct segvn_data *svd = buf;

279         rw_destroy(&svd->lock);
280         mutex_destroy(&svd->segfree_syncmtx);
281 }

283 /*ARGSUSED*/
284 static int
285 svntr_cache_constructor(void *buf, void *cdrarg, int kmflags)
286 {
287         bzero(buf, sizeof (svntr_t));
288         return (0);
289 }

291 /*
292  * Patching this variable to non-zero allows the system to run with
293  * stacks marked as "not executable".  It's a bit of a kludge, but is
294  * provided as a tweakable for platforms that export those ABIs
295  * (e.g. sparc V8) that have executable stacks enabled by default.
296  * There are also some restrictions for platforms that don't actually
297  * implement 'noexec' protections.
298  *
299  * Once enabled, the system is (therefore) unable to provide a fully
300  * ABI-compliant execution environment, though practically speaking,
301  * most everything works.  The exceptions are generally some interpreters
302  * and debuggers that create executable code on the stack and jump
303  * into it (without explicitly mprotecting the address range to include
304  * PROT_EXEC).
305  *
306  * One important class of applications that are disabled are those
307  * that have been transformed into malicious agents using one of the
308  * numerous "buffer overflow" attacks.  See 4007890.
309  */
310 int noexec_user_stack = 0;
311 int noexec_user_stack_log = 1;

313 int segvn_lpg_disable = 0;
314 uint_t segvn_maxpgszc = 0;

316 ulong_t segvn_vmpss_clrszc_cnt;
317 ulong_t segvn_vmpss_clrszc_err;
318 ulong_t segvn_fltvnpages_clrszc_cnt;
319 ulong_t segvn_fltvnpages_clrszc_err;
320 ulong_t segvn_setpgsz_align_err;
321 ulong_t segvn_setpgsz_anon_align_err;
322 ulong_t segvn_setpgsz_getattr_err;
323 ulong_t segvn_setpgsz_eof_err;
324 ulong_t segvn_faultvnmpss_align_err1;
```



```
325 ulong_t segvn_faultvnmpss_align_err2;
326 ulong_t segvn_faultvnmpss_align_err3;
327 ulong_t segvn_faultvnmpss_align_err4;
328 ulong_t segvn_faultvnmpss_align_err5;
329 ulong_t segvn_vmpss_pageio_deadlk_err;

331 int segvn_use_regions = 1;

333 /*
334  * Segvn supports text replication optimization for NUMA platforms. Text
335  * replica's are represented by anon maps (amp). There's one amp per text file
336  * region per lgroup. A process chooses the amp for each of its text mappings
337  * based on the lgroup assignment of its main thread (t_tid = 1). All
338  * processes that want a replica on a particular lgroup for the same text file
339  * mapping share the same amp. amp's are looked up in svntr_hashtab hash table
340  * with vp,off,size,szc used as a key. Text replication segments are read only
341  * MAP_PRIVATE|MAP_TEXT segments that map vnode. Replication is achieved by
342  * forcing COW faults from vnode to amp and mapping amp pages instead of vnode
343  * pages. Replication amp is assigned to a segment when it gets its first
344  * pagefault. To handle main thread lgroup rehoming segvn_trasync_thread
345  * rechecks periodically if the process still maps an amp local to the main
346  * thread. If not async thread forces process to remap to an amp in the new
347  * home lgroup of the main thread. Current text replication implementation
348  * only provides the benefit to workloads that do most of their work in the
349  * main thread of a process or all the threads of a process run in the same
350  * lgroup. To extend text replication benefit to different types of
351  * multithreaded workloads further work would be needed in the hat layer to
352  * allow the same virtual address in the same hat to simultaneously map
353  * different physical addresses (i.e. page table replication would be needed
354  * for x86).
355  *
356  * amp pages are used instead of vnode pages as long as segment has a very
357  * simple life cycle.  It's created via segvn_create(), handles S_EXEC
358  * (S_READ) pagefaults and is fully unmapped.  If anything more complicated
359  * happens such as protection is changed, real COW fault happens, pagesize is
360  * changed, MC_LOCK is requested or segment is partially unmapped we turn off
361  * text replication by converting the segment back to vnode only segment
362  * (unmap segment's address range and set svd->amp to NULL).
363  *
364  * The original file can be changed after amp is inserted into
365  * svntr_hashtab. Processes that are launched after the file is already
366  * changed can't use the replica's created prior to the file change. To
367  * implement this functionality hash entries are timestamped. Replica's can
368  * only be used if current file modification time is the same as the timestamp
369  * saved when hash entry was created. However just timestamps alone are not
370  * sufficient to detect file modification via mmap(MAP_SHARED) mappings. We
371  * deal with file changes via MAP_SHARED mappings differently. When writable
372  * MAP_SHARED mappings are created to vnodes marked as executable we mark all
373  * existing replica's for this vnode as not usable for future text
374  * mappings. And we don't create new replica's for files that currently have
375  * potentially writable MAP_SHARED mappings (i.e. vn_is_mapped(V_WRITE) is
376  * true).
377  */

379 #define SEGVN_TEXTREPL_MAXBYTES_FACTOR  (20)
380 size_t  segvn_textrepl_max_bytes_factor = SEGVN_TEXTREPL_MAXBYTES_FACTOR;

382 static ulong_t                  svntr_hashtab_sz = 512;
383 static svntr_bucket_t           *svntr_hashtab = NULL;
384 static struct kmem_cache        *svntr_cache;
385 static svntr_stats_t            *segvn_textrepl_stats;
386 static ksema_t                  segvn_trasync_sem;

388 int                             segvn_disable_textrepl = 1;
389 size_t                          textrepl_size_thresh = (size_t)-1;
390 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;
                }
```

```
523                 /*
524                  * set v_mpssdata just once per vnode life
525                  * so that it never changes.
526                  */
527                 mutex_enter(&vp->v_lock);
528                 if (vp->v_mpssdata == NULL) {
529                         if (err == EINVAL) {
530                                 vp->v_mpssdata = SEGVN_PAGEIO;
531                         } else {
532                                 vp->v_mpssdata = SEGVN_NOPAGEIO;
533                         }
534                 }
535                 mutex_exit(&vp->v_lock);
536         }
537 }

539 int
540 segvn_create(struct seg *seg, void *argsp)
541 {
542         struct segvn_crargs *a = (struct segvn_crargs *)argsp;
543         struct segvn_data *svd;
544         size_t swresv = 0;
545         struct cred *cred;
546         struct anon_map *amp;
547         int error = 0;
548         size_t pgsz;
549         lgrp_mem_policy_t mpolicy = LGRP_MEM_POLICY_DEFAULT;
550         int use_rgn = 0;
551         int trok = 0;

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

555         if (a->type != MAP_PRIVATE && a->type != MAP_SHARED) {
556                 panic("segvn_create type");
557                 /*NOTREACHED*/
558         }

560         /*
561          * Check arguments.  If a shared anon structure is given then
562          * it is illegal to also specify a vp.
563          */
564         if (a->amp != NULL && a->vp != NULL) {
565                 panic("segvn_create anon_map");
566                 /*NOTREACHED*/
567         }

569         if (a->type == MAP_PRIVATE && (a->flags & MAP_TEXT) &&
570             a->vp != NULL && a->prot == (PROT_USER | PROT_READ | PROT_EXEC) &&
571             segvn_use_regions) {
572                 use_rgn = 1;
573         }

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

579         if (a->szc != 0) {
580                 if (segvn_lpg_disable != 0 || (a->szc == AS_MAP_NO_LPOOB) ||
581                     (a->amp != NULL && a->type == MAP_PRIVATE) ||
582                     (a->flags & MAP_NORESERVE) || seg->s_as == &kas) {
583                         a->szc = 0;
584                 } else {
585                         if (a->szc > segvn_maxpgszc)
586                                 a->szc = segvn_maxpgszc;
587                         pgsz = page_get_pagesize(a->szc);
588                         if (!IS_P2ALIGNED(seg->s_base, pgsz) ||
```



```
589                             !IS_P2ALIGNED(seg->s_size, pgsz)) {
590                                 a->szc = 0;
591                         } else if (a->vp != NULL) {
592                                 if (IS_SWAPFSVP(a->vp) || VN_ISKAS(a->vp)) {
593                                         /*
594                                          * paranoid check.
595                                          * hat_page_demote() is not supported
596                                          * on swapfs pages.
597                                          */
598                                         a->szc = 0;
599                                 } else if (map_addr_vacalign_check(seg->s_base,
600                                     a->offset & PAGEMASK)) {
601                                         a->szc = 0;
602                                 }
603                         } else if (a->amp != NULL) {
604                                 pgcnt_t anum = btopr(a->offset);
605                                 pgcnt_t pgcnt = page_get_pagecnt(a->szc);
606                                 if (!IS_P2ALIGNED(anum, pgcnt)) {
607                                         a->szc = 0;
608                                 }
609                         }
610                 }
611         }

613         /*
614          * If segment may need private pages, reserve them now.
615          */
616         if (!(a->flags & MAP_NORESERVE) && ((a->vp == NULL && a->amp == NULL) ||
617             (a->type == MAP_PRIVATE && (a->prot & PROT_WRITE)))) {
618                 if (anon_resv_zone(seg->s_size,
619                     seg->s_as->a_proc->p_zone) == 0)
620                         return (EAGAIN);
621                 swresv = seg->s_size;
622                 TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
623                     seg, swresv, 1);
624         }

626         /*
627          * Reserve any mapping structures that may be required.
628          *
629          * Don't do it for segments that may use regions. It's currently a
630          * noop in the hat implementations anyway.
631          */
632         if (!use_rgn) {
633                 hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);
634         }

636         if (a->cred) {
637                 cred = a->cred;
638                 crhold(cred);
639         } else {
640                 crhold(cred = CRED());
641         }

643         /* Inform the vnode of the new mapping */
644         if (a->vp != NULL) {
645                 error = VOP_ADDMAP(a->vp, a->offset & PAGEMASK,
646                     seg->s_as, seg->s_base, seg->s_size, a->prot,
647                     a->maxprot, a->type, cred, NULL);
648                 if (error) {
649                         if (swresv != 0) {
650                                 anon_unresv_zone(swresv,
651                                     seg->s_as->a_proc->p_zone);
652                                 TRACE_3(TR_FAC_VM, TR_ANON_PROC,
653                                     "anon proc:%p %lu %u", seg, swresv, 0);
654                         }
```

```
655                         crfree(cred);
656                         if (!use_rgn) {
657                                 hat_unload(seg->s_as->a_hat, seg->s_base,
658                                     seg->s_size, HAT_UNLOAD_UNMAP);
659                         }
660                         return (error);
661                 }
662                 /*
663                  * svntr_hashtab will be NULL if we support shared regions.
664                  */
665                 trok = ((a->flags & MAP_TEXT) &&
666                     (seg->s_size > textrepl_size_thresh ||
667                     (a->flags & _MAP_TEXTREPL)) &&
668                     lgrp_optimizations() && svntr_hashtab != NULL &&
669                     a->type == MAP_PRIVATE && swresv == 0 &&
670                     !(a->flags & MAP_NORESERVE) &&
671                     seg->s_as != &kas && a->vp->v_type == VREG);
672
673                 ASSERT(!trok || !use_rgn);
674         }
675
676         /*
677          * MAP_NORESERVE mappings don't count towards the VSZ of a process
678          * until we fault the pages in.
679          */
680         if ((a->vp == NULL || a->vp->v_type != VREG) &&
681             a->flags & MAP_NORESERVE) {
682                 seg->s_as->a_resvsize -= seg->s_size;
683         }
684
685         /*
686          * If more than one segment in the address space, and they're adjacent
687          * virtually, try to concatenate them.  Don't concatenate if an
688          * explicit anon_map structure was supplied (e.g., SystemV shared
689          * memory) or if we'll use text replication for this segment.
690          */
691         if (a->amp == NULL && !use_rgn && !trok) {
692                 struct seg *pseg, *nseg;
693                 struct segvn_data *psvd, *nsvd;
694                 lgrp_mem_policy_t ppolicy, npolicy;
695                 uint_t  lgrp_mem_policy_flags = 0;
696                 extern lgrp_mem_policy_t lgrp_mem_default_policy;
697
698                 /*
699                  * Memory policy flags (lgrp_mem_policy_flags) is valid when
700                  * extending stack/heap segments.
701                  */
702                 if ((a->vp == NULL) && (a->type == MAP_PRIVATE) &&
703                     !(a->flags & MAP_NORESERVE) && (seg->s_as != &kas)) {
704                         lgrp_mem_policy_flags = a->lgrp_mem_policy_flags;
705                 } else {
706                         /*
707                          * Get policy when not extending it from another segment
708                          */
709                         mpolicy = lgrp_mem_policy_default(seg->s_size, a->type);
710                 }
711
712                 /*
713                  * First, try to concatenate the previous and new segments
714                  */
715                 pseg = AS_SEGPREV(seg->s_as, seg);
716                 if (pseg != NULL &&
717                     pseg->s_base + pseg->s_size == seg->s_base &&
718                     pseg->s_ops == &segvn_ops) {
719                         /*
720                          * Get memory allocation policy from previous segment.
```



```
721                          * When extension is specified (e.g. for heap) apply
722                          * this policy to the new segment regardless of the
723                          * outcome of segment concatenation.  Extension occurs
724                          * for non-default policy otherwise default policy is
725                          * used and is based on extended segment size.
726                          */
727                         psvd = (struct segvn_data *)pseg->s_data;
728                         ppolicy = psvd->policy_info.mem_policy;
729                         if (lgrp_mem_policy_flags ==
730                             LGRP_MP_FLAG_EXTEND_UP) {
731                                 if (ppolicy != lgrp_mem_default_policy) {
732                                         mpolicy = ppolicy;
733                                 } else {
734                                         mpolicy = lgrp_mem_policy_default(
735                                             pseg->s_size + seg->s_size,
736                                             a->type);
737                                 }
738                         }
739
740                         if (mpolicy == ppolicy &&
741                             (pseg->s_size + seg->s_size <=
742                             segvn_comb_thrshld || psvd->amp == NULL) &&
743                             segvn_extend_prev(pseg, seg, a, swresv) == 0) {
744                                 /*
745                                  * success! now try to concatenate
746                                  * with following seg
747                                  */
748                                 crfree(cred);
749                                 nseg = AS_SEGNEXT(pseg->s_as, pseg);
750                                 if (nseg != NULL &&
751                                     nseg != pseg &&
752                                     nseg->s_ops == &segvn_ops &&
753                                     pseg->s_base + pseg->s_size ==
754                                     nseg->s_base)
755                                         (void) segvn_concat(pseg, nseg, 0);
756                                 ASSERT(pseg->s_szc == 0 ||
757                                     (a->szc == pseg->s_szc &&
758                                     IS_P2ALIGNED(pseg->s_base, pgsz) &&
759                                     IS_P2ALIGNED(pseg->s_size, pgsz)));
760                                 return (0);
761                         }
762                 }
763
764                 /*
765                  * Failed, so try to concatenate with following seg
766                  */
767                 nseg = AS_SEGNEXT(seg->s_as, seg);
768                 if (nseg != NULL &&
769                     seg->s_base + seg->s_size == nseg->s_base &&
770                     nseg->s_ops == &segvn_ops) {
771                         /*
772                          * Get memory allocation policy from next segment.
773                          * When extension is specified (e.g. for stack) apply
774                          * this policy to the new segment regardless of the
775                          * outcome of segment concatenation.  Extension occurs
776                          * for non-default policy otherwise default policy is
777                          * used and is based on extended segment size.
778                          */
779                         nsvd = (struct segvn_data *)nseg->s_data;
780                         npolicy = nsvd->policy_info.mem_policy;
781                         if (lgrp_mem_policy_flags ==
782                             LGRP_MP_FLAG_EXTEND_DOWN) {
783                                 if (npolicy != lgrp_mem_default_policy) {
784                                         mpolicy = npolicy;
785                                 } else {
786                                         mpolicy = lgrp_mem_policy_default(
```

```
787                                             nseg->s_size + seg->s_size,
788                                             a->type);
789                                 }
790                         }
791
792                         if (mpolicy == npolicy &&
793                             segvn_extend_next(seg, nseg, a, swresv) == 0) {
794                                 crfree(cred);
795                                 ASSERT(nseg->s_szc == 0 ||
796                                     (a->szc == nseg->s_szc &&
797                                     IS_P2ALIGNED(nseg->s_base, pgsz) &&
798                                     IS_P2ALIGNED(nseg->s_size, pgsz)));
799                                 return (0);
800                         }
801                 }
802         }
803
804         if (a->vp != NULL) {
805                 VN_HOLD(a->vp);
806                 if (a->type == MAP_SHARED)
807                         lgrp_shm_policy_init(NULL, a->vp);
808         }
809         svd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
810
811         seg->s_ops = &segvn_ops;
812         seg->s_data = (void *)svd;
813         seg->s_szc = a->szc;
814
815         svd->seg = seg;
816         svd->vp = a->vp;
817         /*
818          * Anonymous mappings have no backing file so the offset is meaningless.
819          */
820         svd->offset = a->vp ? (a->offset & PAGEMASK) : 0;
821         svd->prot = a->prot;
822         svd->maxprot = a->maxprot;
823         svd->pageprot = 0;
824         svd->type = a->type;
825         svd->vpage = NULL;
826         svd->cred = cred;
827         svd->advice = MADV_NORMAL;
828         svd->pageadvice = 0;
829         svd->flags = (ushort_t)a->flags;
830         svd->softlockcnt = 0;
831         svd->softlockcnt_sbase = 0;
832         svd->softlockcnt_send = 0;
833         svd->rcookie = HAT_INVALID_REGION_COOKIE;
834         svd->pageswap = 0;
835
836         if (a->szc != 0 && a->vp != NULL) {
837                 segvn_setvnode_mpss(a->vp);
838         }
839         if (svd->type == MAP_SHARED && svd->vp != NULL &&
840             (svd->vp->v_flag & VVMEXEC) && (svd->prot & PROT_WRITE)) {
841                 ASSERT(vn_is_mapped(svd->vp, V_WRITE));
842                 segvn_inval_trcache(svd->vp);
843         }
844
845         amp = a->amp;
846         if ((svd->amp = amp) == NULL) {
847                 svd->anon_index = 0;
848                 if (svd->type == MAP_SHARED) {
849                         svd->swresv = 0;
850                         /*
851                          * Shared mappings to a vp need no other setup.
852                          * If we have a shared mapping to an anon_map object
```



```
853                          * which hasn't been allocated yet,  allocate the
854                          * struct now so that it will be properly shared
855                          * by remembering the swap reservation there.
856                          */
857                         if (a->vp == NULL) {
858                                 svd->amp = anonmap_alloc(seg->s_size, swresv,
859                                     ANON_SLEEP);
860                                 svd->amp->a_szc = seg->s_szc;
861                         }
862                 } else {
863                         /*
864                          * Private mapping (with or without a vp).
865                          * Allocate anon_map when needed.
866                          */
867                         svd->swresv = swresv;
868                 }
869         } else {
870                 pgcnt_t anon_num;
871
872                 /*
873                  * Mapping to an existing anon_map structure without a vp.
874                  * For now we will insure that the segment size isn't larger
875                  * than the size - offset gives us.  Later on we may wish to
876                  * have the anon array dynamically allocated itself so that
877                  * we don't always have to allocate all the anon pointer slots.
878                  * This of course involves adding extra code to check that we
879                  * aren't trying to use an anon pointer slot beyond the end
880                  * of the currently allocated anon array.
881                  */
882                 if ((amp->size - a->offset) < seg->s_size) {
883                         panic("segvn_create anon_map size");
884                         /*NOTREACHED*/
885                 }
886
887                 anon_num = btopr(a->offset);
888
889                 if (a->type == MAP_SHARED) {
890                         /*
891                          * SHARED mapping to a given anon_map.
892                          */
893                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
894                         amp->refcnt++;
895                         if (a->szc > amp->a_szc) {
896                                 amp->a_szc = a->szc;
897                         }
898                         ANON_LOCK_EXIT(&amp->a_rwlock);
899                         svd->anon_index = anon_num;
900                         svd->swresv = 0;
901                 } else {
902                         /*
903                          * PRIVATE mapping to a given anon_map.
904                          * Make sure that all the needed anon
905                          * structures are created (so that we will
906                          * share the underlying pages if nothing
907                          * is written by this mapping) and then
908                          * duplicate the anon array as is done
909                          * when a privately mapped segment is dup'ed.
910                          */
911                         struct anon *ap;
912                         caddr_t addr;
913                         caddr_t eaddr;
914                         ulong_t anon_idx;
915                         int hat_flag = HAT_LOAD;
916
917                         if (svd->flags & MAP_TEXT) {
918                                 hat_flag |= HAT_LOAD_TEXT;
```

```
919                         }
921                         svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
922                         svd->amp->a_szc = seg->s_szc;
923                         svd->anon_index = 0;
924                         svd->swresv = swresv;
926                         /*
927                          * Prevent 2 threads from allocating anon
928                          * slots simultaneously.
929                          */
930                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
931                         eaddr = seg->s_base + seg->s_size;
933                         for (anon_idx = anon_num, addr = seg->s_base;
934                             addr < eaddr; addr += PAGESIZE, anon_idx++) {
935                             page_t *pp;
937                             if ((ap = anon_get_ptr(amp->ahp,
938                                 anon_idx)) != NULL)
939                                     continue;
941                             /*
942                              * Allocate the anon struct now.
943                              * Might as well load up translation
944                              * to the page while we're at it...
945                              */
946                             pp = anon_zero(seg, addr, &ap, cred);
947                             if (ap == NULL || pp == NULL) {
948                                     panic("segvn_create anon_zero");
949                                     /*NOTREACHED*/
950                             }
952                             /*
953                              * Re-acquire the anon_map lock and
954                              * initialize the anon array entry.
955                              */
956                             ASSERT(anon_get_ptr(amp->ahp,
957                                 anon_idx) == NULL);
958                             (void) anon_set_ptr(amp->ahp, anon_idx, ap,
959                                 ANON_SLEEP);
961                             ASSERT(seg->s_szc == 0);
962                             ASSERT(!IS_VMODSORT(pp->p_vnode));
964                             ASSERT(use_rgn == 0);
965                             hat_memload(seg->s_as->a_hat, addr, pp,
966                                 svd->prot & ~PROT_WRITE, hat_flag);
968                             page_unlock(pp);
969                         }
970                         ASSERT(seg->s_szc == 0);
971                         anon_dup(amp->ahp, anon_num, svd->amp->ahp,
972                             0, seg->s_size);
973                         ANON_LOCK_EXIT(&amp->a_rwlock);
974                 }
975         }
977         /*
978          * Set default memory allocation policy for segment
979          *
980          * Always set policy for private memory at least for initialization
981          * even if this is a shared memory segment
982          */
983         (void) lgrp_privm_policy_set(mpolicy, &svd->policy_info, seg->s_size);
```



```
985         if (svd->type == MAP_SHARED)
986                 (void) lgrp_shm_policy_set(mpolicy, svd->amp, svd->anon_index,
987                     svd->vp, svd->offset, seg->s_size);
989         if (use_rgn) {
990                 ASSERT(!trok);
991                 ASSERT(svd->amp == NULL);
992                 svd->rcookie = hat_join_region(seg->s_as->a_hat, seg->s_base,
993                     seg->s_size, (void *)svd->vp, svd->offset, svd->prot,
994                     (uchar_t)seg->s_szc, segvn_hat_rgn_unload_callback,
995                     HAT_REGION_TEXT);
996         }
998         ASSERT(!trok || !(svd->prot & PROT_WRITE));
999         svd->tr_state = trok ? SEGVN_TR_INIT : SEGVN_TR_OFF;
1001        return (0);
1002 }
1004 /*
1005  * Concatenate two existing segments, if possible.
1006  * Return 0 on success, -1 if two segments are not compatible
1007  * or -2 on memory allocation failure.
1008  * If amp_cat == 1 then try and concat segments with anon maps
1009  */
1010 static int
1011 segvn_concat(struct seg *seg1, struct seg *seg2, int amp_cat)
1012 {
1013         struct segvn_data *svd1 = seg1->s_data;
1014         struct segvn_data *svd2 = seg2->s_data;
1015         struct anon_map *amp1 = svd1->amp;
1016         struct anon_map *amp2 = svd2->amp;
1017         struct vpage *vpage1 = svd1->vpage;
1018         struct vpage *vpage2 = svd2->vpage, *nvpage = NULL;
1019         size_t size, nvpsize;
1020         pgcnt_t npages1, npages2;
1022         ASSERT(seg1->s_as && seg2->s_as && seg1->s_as == seg2->s_as);
1023         ASSERT(AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));
1024         ASSERT(seg1->s_ops == seg2->s_ops);
1026         if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie) ||
1027             HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1028                 return (-1);
1029         }
1031         /* both segments exist, try to merge them */
1032 #define incompat(x)     (svd1->x != svd2->x)
1033         if (incompat(vp) || incompat(maxprot) ||
1034             (!svd1->pageadvice && !svd2->pageadvice && incompat(advice)) ||
1035             (!svd1->pageprot && !svd2->pageprot && incompat(prot)) ||
1036             incompat(type) || incompat(cred) || incompat(flags) ||
1037             seg1->s_szc != seg2->s_szc || incompat(policy_info.mem_policy) ||
1038             (svd2->softlockcnt > 0) || svd1->softlockcnt_send > 0)
1039                 return (-1);
1040 #undef incompat
1042         /*
1043          * vp == NULL implies zfod, offset doesn't matter
1044          */
1045         if (svd1->vp != NULL &&
1046             svd1->offset + seg1->s_size != svd2->offset) {
1047                 return (-1);
1048         }
1050         /*
```

```
1051          * Don't concatenate if either segment uses text replication.
1052          */
1053         if (svd1->tr_state != SEGVN_TR_OFF || svd2->tr_state != SEGVN_TR_OFF) {
1054                 return (-1);
1055         }
1056
1057         /*
1058          * Fail early if we're not supposed to concatenate
1059          * segments with non NULL amp.
1060          */
1061         if (amp_cat == 0 && (amp1 != NULL || amp2 != NULL)) {
1062                 return (-1);
1063         }
1064
1065         if (svd1->vp == NULL && svd1->type == MAP_SHARED) {
1066                 if (amp1 != amp2) {
1067                         return (-1);
1068                 }
1069                 if (amp1 != NULL && svd1->anon_index + btop(seg1->s_size) !=
1070                     svd2->anon_index) {
1071                         return (-1);
1072                 }
1073                 ASSERT(amp1 == NULL || amp1->refcnt >= 2);
1074         }
1075
1076         /*
1077          * If either seg has vpages, create a new merged vpage array.
1078          */
1079         if (vpage1 != NULL || vpage2 != NULL) {
1080                 struct vpage *vp, *evp;
1081
1082                 npages1 = seg_pages(seg1);
1083                 npages2 = seg_pages(seg2);
1084                 nvpsize = vpgtob(npages1 + npages2);
1085
1086                 if ((nvpage = kmem_zalloc(nvpsize, KM_NOSLEEP)) == NULL) {
1087                         return (-2);
1088                 }
1089
1090                 if (vpage1 != NULL) {
1091                         bcopy(vpage1, nvpage, vpgtob(npages1));
1092                 } else {
1093                         evp = nvpage + npages1;
1094                         for (vp = nvpage; vp < evp; vp++) {
1095                                 VPP_SETPROT(vp, svd1->prot);
1096                                 VPP_SETADVICE(vp, svd1->advice);
1097                         }
1098                 }
1099
1100                 if (vpage2 != NULL) {
1101                         bcopy(vpage2, nvpage + npages1, vpgtob(npages2));
1102                 } else {
1103                         evp = nvpage + npages1 + npages2;
1104                         for (vp = nvpage + npages1; vp < evp; vp++) {
1105                                 VPP_SETPROT(vp, svd2->prot);
1106                                 VPP_SETADVICE(vp, svd2->advice);
1107                         }
1108                 }
1109
1110                 if (svd2->pageswap && (!svd1->pageswap && svd1->swresv)) {
1111                         ASSERT(svd1->swresv == seg1->s_size);
1112                         ASSERT(!(svd1->flags & MAP_NORESERVE));
1113                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1114                         evp = nvpage + npages1;
1115                         for (vp = nvpage; vp < evp; vp++) {
1116                                 VPP_SETSWAPRES(vp);
```



```
1117                         }
1118                 }
1119
1120                 if (svd1->pageswap && (!svd2->pageswap && svd2->swresv)) {
1121                         ASSERT(svd2->swresv == seg2->s_size);
1122                         ASSERT(!(svd1->flags & MAP_NORESERVE));
1123                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1124                         vp = nvpage + npages1;
1125                         evp = vp + npages2;
1126                         for (; vp < evp; vp++) {
1127                                 VPP_SETSWAPRES(vp);
1128                         }
1129                 }
1130         }
1131         ASSERT((vpage1 != NULL || vpage2 != NULL) ||
1132             (svd1->pageswap == 0 && svd2->pageswap == 0));
1133
1134         /*
1135          * If either segment has private pages, create a new merged anon
1136          * array. If mergeing shared anon segments just decrement anon map's
1137          * refcnt.
1138          */
1139         if (amp1 != NULL && svd1->type == MAP_SHARED) {
1140                 ASSERT(amp1 == amp2 && svd1->vp == NULL);
1141                 ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1142                 ASSERT(amp1->refcnt >= 2);
1143                 amp1->refcnt--;
1144                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1145                 svd2->amp = NULL;
1146         } else if (amp1 != NULL || amp2 != NULL) {
1147                 struct anon_hdr *nahp;
1148                 struct anon_map *namp = NULL;
1149                 size_t asize;
1150
1151                 ASSERT(svd1->type == MAP_PRIVATE);
1152
1153                 asize = seg1->s_size + seg2->s_size;
1154                 if ((nahp = anon_create(btop(asize), ANON_NOSLEEP)) == NULL) {
1155                         if (nvpage != NULL) {
1156                                 kmem_free(nvpage, nvpsize);
1157                         }
1158                         return (-2);
1159                 }
1160                 if (amp1 != NULL) {
1161                         /*
1162                          * XXX anon rwlock is not really needed because
1163                          * this is a private segment and we are writers.
1164                          */
1165                         ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1166                         ASSERT(amp1->refcnt == 1);
1167                         if (anon_copy_ptr(amp1->ahp, svd1->anon_index,
1168                             nahp, 0, btop(seg1->s_size), ANON_NOSLEEP)) {
1169                                 anon_release(nahp, btop(asize));
1170                                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1171                                 if (nvpage != NULL) {
1172                                         kmem_free(nvpage, nvpsize);
1173                                 }
1174                                 return (-2);
1175                         }
1176                 }
1177                 if (amp2 != NULL) {
1178                         ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
1179                         ASSERT(amp2->refcnt == 1);
1180                         if (anon_copy_ptr(amp2->ahp, svd2->anon_index,
1181                             nahp, btop(seg1->s_size), btop(seg2->s_size),
1182                             ANON_NOSLEEP)) {
```

```
1183                                anon_release(nahp, btop(asize));
1184                                ANON_LOCK_EXIT(&amp2->a_rwlock);
1185                                if (amp1 != NULL) {
1186                                        ANON_LOCK_EXIT(&amp1->a_rwlock);
1187                                }
1188                                if (nvpage != NULL) {
1189                                        kmem_free(nvpage, nvpsize);
1190                                }
1191                                return (-2);
1192                        }
1193                }
1194                if (amp1 != NULL) {
1195                        namp = amp1;
1196                        anon_release(amp1->ahp, btop(amp1->size));
1197                }
1198                if (amp2 != NULL) {
1199                        if (namp == NULL) {
1200                                ASSERT(amp1 == NULL);
1201                                namp = amp2;
1202                                anon_release(amp2->ahp, btop(amp2->size));
1203                        } else {
1204                                amp2->refcnt--;
1205                                ANON_LOCK_EXIT(&amp2->a_rwlock);
1206                                anonmap_free(amp2);
1207                        }
1208                        svd2->amp = NULL; /* needed for seg_free */
1209                }
1210                namp->ahp = nahp;
1211                namp->size = asize;
1212                svd1->amp = namp;
1213                svd1->anon_index = 0;
1214                ANON_LOCK_EXIT(&namp->a_rwlock);
1215        }
1216        /*
1217         * Now free the old vpage structures.
1218         */
1219        if (nvpage != NULL) {
1220                if (vpage1 != NULL) {
1221                        kmem_free(vpage1, vpgtob(npages1));
1222                }
1223                if (vpage2 != NULL) {
1224                        svd2->vpage = NULL;
1225                        kmem_free(vpage2, vpgtob(npages2));
1226                }
1227                if (svd2->pageprot) {
1228                        svd1->pageprot = 1;
1229                }
1230                if (svd2->pageadvice) {
1231                        svd1->pageadvice = 1;
1232                }
1233                if (svd2->pageswap) {
1234                        svd1->pageswap = 1;
1235                }
1236                svd1->vpage = nvpage;
1237        }

1239        /* all looks ok, merge segments */
1240        svd1->swresv += svd2->swresv;
1241        svd2->swresv = 0;  /* so seg_free doesn't release swap space */
1242        size = seg2->s_size;
1243        seg_free(seg2);
1244        seg1->s_size += size;
1245        return (0);
1246 }

1248 /*
```



```
1249  * Extend the previous segment (seg1) to include the
1250  * new segment (seg2 + a), if possible.
1251  * Return 0 on success.
1252  */
1253 static int
1254 segvn_extend_prev(seg1, seg2, a, swresv)
1255         struct seg *seg1, *seg2;
1256         struct segvn_crargs *a;
1257         size_t swresv;
1258 {
1259         struct segvn_data *svd1 = (struct segvn_data *)seg1->s_data;
1260         size_t size;
1261         struct anon_map *amp1;
1262         struct vpage *new_vpage;

1264         /*
1265          * We don't need any segment level locks for "segvn" data
1266          * since the address space is "write" locked.
1267          */
1268         ASSERT(seg1->s_as && AS_WRITE_HELD(seg1->s_as, &seg1->s_as->a_lock));

1270         if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie)) {
1271                 return (-1);
1272         }

1274         /* second segment is new, try to extend first */
1275         /* XXX - should also check cred */
1276         if (svd1->vp != a->vp || svd1->maxprot != a->maxprot ||
1277             (!svd1->pageprot && (svd1->prot != a->prot)) ||
1278             svd1->type != a->type || svd1->flags != a->flags ||
1279             seg1->s_szc != a->szc || svd1->softlockcnt_send > 0)
1280                 return (-1);

1282         /* vp == NULL implies zfod, offset doesn't matter */
1283         if (svd1->vp != NULL &&
1284             svd1->offset + seg1->s_size != (a->offset & PAGEMASK))
1285                 return (-1);

1287         if (svd1->tr_state != SEGVN_TR_OFF) {
1288                 return (-1);
1289         }

1291         amp1 = svd1->amp;
1292         if (amp1) {
1293                 pgcnt_t newpgs;

1295                 /*
1296                  * Segment has private pages, can data structures
1297                  * be expanded?
1298                  *
1299                  * Acquire the anon_map lock to prevent it from changing,
1300                  * if it is shared.  This ensures that the anon_map
1301                  * will not change while a thread which has a read/write
1302                  * lock on an address space references it.
1303                  * XXX - Don't need the anon_map lock at all if "refcnt"
1304                  * is 1.
1305                  *
1306                  * Can't grow a MAP_SHARED segment with an anonmap because
1307                  * there may be existing anon slots where we want to extend
1308                  * the segment and we wouldn't know what to do with them
1309                  * (e.g., for tmpfs right thing is to just leave them there,
1310                  * for /dev/zero they should be cleared out).
1311                  */
1312                 if (svd1->type == MAP_SHARED)
1313                         return (-1);
```

```
1315                ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1316                if (amp1->refcnt > 1) {
1317                        ANON_LOCK_EXIT(&amp1->a_rwlock);
1318                        return (-1);
1319                }
1320                newpgs = anon_grow(amp1->ahp, &svd1->anon_index,
1321                    btop(seg1->s_size), btop(seg2->s_size), ANON_NOSLEEP);

1323                if (newpgs == 0) {
1324                        ANON_LOCK_EXIT(&amp1->a_rwlock);
1325                        return (-1);
1326                }
1327                amp1->size = ptob(newpgs);
1328                ANON_LOCK_EXIT(&amp1->a_rwlock);
1329        }
1330        if (svd1->vpage != NULL) {
1331                struct vpage *vp, *evp;
1332                new_vpage =
1333                    kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
1334                        KM_NOSLEEP);
1335                if (new_vpage == NULL)
1336                        return (-1);
1337                bcopy(svd1->vpage, new_vpage, vpgtob(seg_pages(seg1)));
1338                kmem_free(svd1->vpage, vpgtob(seg_pages(seg1)));
1339                svd1->vpage = new_vpage;

1341                vp = new_vpage + seg_pages(seg1);
1342                evp = vp + seg_pages(seg2);
1343                for (; vp < evp; vp++)
1344                        VPP_SETPROT(vp, a->prot);
1345                if (svd1->pageswap && swresv) {
1346                        ASSERT(!(svd1->flags & MAP_NORESERVE));
1347                        ASSERT(swresv == seg2->s_size);
1348                        vp = new_vpage + seg_pages(seg1);
1349                        for (; vp < evp; vp++) {
1350                                VPP_SETSWAPRES(vp);
1351                        }
1352                }
1353        }
1354        ASSERT(svd1->vpage != NULL || svd1->pageswap == 0);
1355        size = seg2->s_size;
1356        seg_free(seg2);
1357        seg1->s_size += size;
1358        svd1->swresv += swresv;
1359        if (svd1->pageprot && (a->prot & PROT_WRITE) &&
1360            svd1->type == MAP_SHARED && svd1->vp != NULL &&
1361            (svd1->vp->v_flag & VVMEXEC)) {
1362                ASSERT(vn_is_mapped(svd1->vp, V_WRITE));
1363                segvn_inval_trcache(svd1->vp);
1364        }
1365        return (0);
1366 }

1368 /*
1369  * Extend the next segment (seg2) to include the
1370  * new segment (seg1 + a), if possible.
1371  * Return 0 on success.
1372  */
1373 static int
1374 segvn_extend_next(
1375        struct seg *seg1,
1376        struct seg *seg2,
1377        struct segvn_crargs *a,
1378        size_t swresv)
1379 {
1380        struct segvn_data *svd2 = (struct segvn_data *)seg2->s_data;
```



```
1381        size_t size;
1382        struct anon_map *amp2;
1383        struct vpage *new_vpage;

1385        /*
1386         * We don't need any segment level locks for "segvn" data
1387         * since the address space is "write" locked.
1388         */
1389        ASSERT(seg2->s_as && AS_WRITE_HELD(seg2->s_as, &seg2->s_as->a_lock));

1391        if (HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1392                return (-1);
1393        }

1395        /* first segment is new, try to extend second */
1396        /* XXX - should also check cred */
1397        if (svd2->vp != a->vp || svd2->maxprot != a->maxprot ||
1398            (!svd2->pageprot && (svd2->prot != a->prot)) ||
1399            svd2->type != a->type || svd2->flags != a->flags ||
1400            seg2->s_szc != a->szc || svd2->softlockcnt_sbase > 0)
1401                return (-1);
1402        /* vp == NULL implies zfod, offset doesn't matter */
1403        if (svd2->vp != NULL &&
1404            (a->offset & PAGEMASK) + seg1->s_size != svd2->offset)
1405                return (-1);

1407        if (svd2->tr_state != SEGVN_TR_OFF) {
1408                return (-1);
1409        }

1411        amp2 = svd2->amp;
1412        if (amp2) {
1413                pgcnt_t newpgs;

1415                /*
1416                 * Segment has private pages, can data structures
1417                 * be expanded?
1418                 *
1419                 * Acquire the anon_map lock to prevent it from changing,
1420                 * if it is shared.  This ensures that the anon_map
1421                 * will not change while a thread which has a read/write
1422                 * lock on an address space references it.
1423                 *
1424                 * XXX - Don't need the anon_map lock at all if "refcnt"
1425                 * is 1.
1426                 */
1427                if (svd2->type == MAP_SHARED)
1428                        return (-1);

1430                ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
1431                if (amp2->refcnt > 1) {
1432                        ANON_LOCK_EXIT(&amp2->a_rwlock);
1433                        return (-1);
1434                }
1435                newpgs = anon_grow(amp2->ahp, &svd2->anon_index,
1436                    btop(seg2->s_size), btop(seg1->s_size),
1437                    ANON_NOSLEEP | ANON_GROWDOWN);

1439                if (newpgs == 0) {
1440                        ANON_LOCK_EXIT(&amp2->a_rwlock);
1441                        return (-1);
1442                }
1443                amp2->size = ptob(newpgs);
1444                ANON_LOCK_EXIT(&amp2->a_rwlock);
1445        }
1446        if (svd2->vpage != NULL) {
```

```
1447                 struct vpage *vp, *evp;
1448                 new_vpage =
1449                     kmem_zalloc(vpgtob(seg_pages(seg1) + seg_pages(seg2)),
1450                     KM_NOSLEEP);
1451                 if (new_vpage == NULL) {
1452                         /* Not merging segments so adjust anon_index back */
1453                         if (amp2)
1454                                 svd2->anon_index += seg_pages(seg1);
1455                         return (-1);
1456                 }
1457                 bcopy(svd2->vpage, new_vpage + seg_pages(seg1),
1458                     vpgtob(seg_pages(seg2)));
1459                 kmem_free(svd2->vpage, vpgtob(seg_pages(seg2)));
1460                 svd2->vpage = new_vpage;

1462                 vp = new_vpage;
1463                 evp = vp + seg_pages(seg1);
1464                 for (; vp < evp; vp++)
1465                         VPP_SETPROT(vp, a->prot);
1466                 if (svd2->pageswap && swresv) {
1467                         ASSERT(!(svd2->flags & MAP_NORESERVE));
1468                         ASSERT(swresv == seg1->s_size);
1469                         vp = new_vpage;
1470                         for (; vp < evp; vp++) {
1471                                 VPP_SETSWAPRES(vp);
1472                         }
1473                 }
1474         }
1475         ASSERT(svd2->vpage != NULL || svd2->pageswap == 0);
1476         size = seg1->s_size;
1477         seg_free(seg1);
1478         seg2->s_size += size;
1479         seg2->s_base -= size;
1480         svd2->offset -= size;
1481         svd2->swresv += swresv;
1482         if (svd2->pageprot && (a->prot & PROT_WRITE) &&
1483             svd2->type == MAP_SHARED && svd2->vp != NULL &&
1484             (svd2->vp->v_flag & VVMEXEC)) {
1485                 ASSERT(vn_is_mapped(svd2->vp, V_WRITE));
1486                 segvn_inval_trcache(svd2->vp);
1487         }
1488         return (0);
1489 }

1491 static int
1492 segvn_dup(struct seg *seg, struct seg *newseg)
1493 {
1494         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1495         struct segvn_data *newsvd;
1496         pgcnt_t npages = seg_pages(seg);
1497         int error = 0;
1498         uint_t prot;
1499         size_t len;
1500         struct anon_map *amp;

1502         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
1503         ASSERT(newseg->s_as->a_proc->p_parent == curproc);

1505         /*
1506          * If segment has anon reserved, reserve more for the new seg.
1507          * For a MAP_NORESERVE segment swresv will be a count of all the
1508          * allocated anon slots; thus we reserve for the child as many slots
1509          * as the parent has allocated. This semantic prevents the child or
1510          * parent from dieing during a copy-on-write fault caused by trying
1511          * to write a shared pre-existing anon page.
1512          */
```



```
1513         if ((len = svd->swresv) != 0) {
1514                 if (anon_resv(svd->swresv) == 0)
1515                         return (ENOMEM);

1517                 TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
1518                     seg, len, 0);
1519         }

1521         newsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);

1523         newseg->s_ops = &segvn_ops;
1524         newseg->s_data = (void *)newsvd;
1525         newseg->s_szc = seg->s_szc;

1527         newsvd->seg = newseg;
1528         if ((newsvd->vp = svd->vp) != NULL) {
1529                 VN_HOLD(svd->vp);
1530                 if (svd->type == MAP_SHARED)
1531                         lgrp_shm_policy_init(NULL, svd->vp);
1532         }
1533         newsvd->offset = svd->offset;
1534         newsvd->prot = svd->prot;
1535         newsvd->maxprot = svd->maxprot;
1536         newsvd->pageprot = svd->pageprot;
1537         newsvd->type = svd->type;
1538         newsvd->cred = svd->cred;
1539         crhold(newsvd->cred);
1540         newsvd->advice = svd->advice;
1541         newsvd->pageadvice = svd->pageadvice;
1542         newsvd->swresv = svd->swresv;
1543         newsvd->pageswap = svd->pageswap;
1544         newsvd->flags = svd->flags;
1545         newsvd->softlockcnt = 0;
1546         newsvd->softlockcnt_sbase = 0;
1547         newsvd->softlockcnt_send = 0;
1548         newsvd->policy_info = svd->policy_info;
1549         newsvd->rcookie = HAT_INVALID_REGION_COOKIE;

1551         if ((amp = svd->amp) == NULL || svd->tr_state == SEGVN_TR_ON) {
1552                 /*
1553                  * Not attaching to a shared anon object.
1554                  */
1555                 ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie) ||
1556                     svd->tr_state == SEGVN_TR_OFF);
1557                 if (svd->tr_state == SEGVN_TR_ON) {
1558                         ASSERT(newsvd->vp != NULL && amp != NULL);
1559                         newsvd->tr_state = SEGVN_TR_INIT;
1560                 } else {
1561                         newsvd->tr_state = svd->tr_state;
1562                 }
1563                 newsvd->amp = NULL;
1564                 newsvd->anon_index = 0;
1565         } else {
1566                 /* regions for now are only used on pure vnode segments */
1567                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
1568                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1569                 newsvd->tr_state = SEGVN_TR_OFF;
1570                 if (svd->type == MAP_SHARED) {
1571                         newsvd->amp = amp;
1572                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
1573                         amp->refcnt++;
1574                         ANON_LOCK_EXIT(&amp->a_rwlock);
1575                         newsvd->anon_index = svd->anon_index;
1576                 } else {
1577                         int reclaim = 1;
```

```
1579                        /*
1580                         * Allocate and initialize new anon_map structure.
1581                         */
1582                        newsvd->amp = anonmap_alloc(newseg->s_size, 0,
1583                            ANON_SLEEP);
1584                        newsvd->amp->a_szc = newseg->s_szc;
1585                        newsvd->anon_index = 0;

1587                        /*
1588                         * We don't have to acquire the anon_map lock
1589                         * for the new segment (since it belongs to an
1590                         * address space that is still not associated
1591                         * with any process), or the segment in the old
1592                         * address space (since all threads in it
1593                         * are stopped while duplicating the address space).
1594                         */

1596                        /*
1597                         * The goal of the following code is to make sure that
1598                         * softlocked pages do not end up as copy on write
1599                         * pages.  This would cause problems where one
1600                         * thread writes to a page that is COW and a different
1601                         * thread in the same process has softlocked it.  The
1602                         * softlock lock would move away from this process
1603                         * because the write would cause this process to get
1604                         * a copy (without the softlock).
1605                         *
1606                         * The strategy here is to just break the
1607                         * sharing on pages that could possibly be
1608                         * softlocked.
1609                         */
1610 retry:
1611                        if (svd->softlockcnt) {
1612                                struct anon *ap, *newap;
1613                                size_t i;
1614                                uint_t vpprot;
1615                                page_t *anon_pl[1+1], *pp;
1616                                caddr_t addr;
1617                                ulong_t old_idx = svd->anon_index;
1618                                ulong_t new_idx = 0;

1620                                /*
1621                                 * The softlock count might be non zero
1622                                 * because some pages are still stuck in the
1623                                 * cache for lazy reclaim. Flush the cache
1624                                 * now. This should drop the count to zero.
1625                                 * [or there is really I/O going on to these
1626                                 * pages]. Note, we have the writers lock so
1627                                 * nothing gets inserted during the flush.
1628                                 */
1629                                if (reclaim == 1) {
1630                                        segvn_purge(seg);
1631                                        reclaim = 0;
1632                                        goto retry;
1633                                }
1634                                i = btopr(seg->s_size);
1635                                addr = seg->s_base;
1636                                /*
1637                                 * XXX break cow sharing using PAGESIZE
1638                                 * pages. They will be relocated into larger
1639                                 * pages at fault time.
1640                                 */
1641                                while (i-- > 0) {
1642                                        if (ap = anon_get_ptr(amp->ahp,
1643                                            old_idx)) {
1644                                                error = anon_getpage(&ap,
```



```
1645                                                    &vpprot, anon_pl, PAGESIZE,
1646                                                    seg, addr, S_READ,
1647                                                    svd->cred);
1648                                                if (error) {
1649                                                        newsvd->vpage = NULL;
1650                                                        goto out;
1651                                                }
1652                                                /*
1653                                                 * prot need not be computed
1654                                                 * below 'cause anon_private is
1655                                                 * going to ignore it anyway
1656                                                 * as child doesn't inherit
1657                                                 * pagelock from parent.
1658                                                 */
1659                                                prot = svd->pageprot ?
1660                                                    VPP_PROT(
1661                                                    &svd->vpage[
1662                                                    seg_page(seg, addr)])
1663                                                    : svd->prot;
1664                                                pp = anon_private(&newap,
1665                                                    newseg, addr, prot,
1666                                                    anon_pl[0], 0,
1667                                                    newsvd->cred);
1668                                                if (pp == NULL) {
1669                                                        /* no mem abort */
1670                                                        newsvd->vpage = NULL;
1671                                                        error = ENOMEM;
1672                                                        goto out;
1673                                                }
1674                                                (void) anon_set_ptr(
1675                                                    newsvd->amp->ahp, new_idx,
1676                                                    newap, ANON_SLEEP);
1677                                                page_unlock(pp);
1678                                        }
1679                                        addr += PAGESIZE;
1680                                        old_idx++;
1681                                        new_idx++;
1682                                }
1683                        } else {        /* common case */
1684                                if (seg->s_szc != 0) {
1685                                        /*
1686                                         * If at least one of anon slots of a
1687                                         * large page exists then make sure
1688                                         * all anon slots of a large page
1689                                         * exist to avoid partial cow sharing
1690                                         * of a large page in the future.
1691                                         */
1692                                        anon_dup_fill_holes(amp->ahp,
1693                                            svd->anon_index, newsvd->amp->ahp,
1694                                            0, seg->s_size, seg->s_szc,
1695                                            svd->vp != NULL);
1696                                } else {
1697                                        anon_dup(amp->ahp, svd->anon_index,
1698                                            newsvd->amp->ahp, 0, seg->s_size);
1699                                }

1701                                hat_clrattr(seg->s_as->a_hat, seg->s_base,
1702                                    seg->s_size, PROT_WRITE);
1703                        }
1704                }
1705        }
1706        /*
1707         * If necessary, create a vpage structure for the new segment.
1708         * Do not copy any page lock indications.
1709         */
1710        if (svd->vpage != NULL) {
```

```
1711                uint_t i;
1712                struct vpage *ovp = svd->vpage;
1713                struct vpage *nvp;

1715                nvp = newsvd->vpage =
1716                    kmem_alloc(vpgtob(npages), KM_SLEEP);
1717                for (i = 0; i < npages; i++) {
1718                        *nvp = *ovp++;
1719                        VPP_CLRPPLOCK(nvp++);
1720                }
1721        } else
1722                newsvd->vpage = NULL;

1724        /* Inform the vnode of the new mapping */
1725        if (newsvd->vp != NULL) {
1726                error = VOP_ADDMAP(newsvd->vp, (offset_t)newsvd->offset,
1727                    newseg->s_as, newseg->s_base, newseg->s_size, newsvd->prot,
1728                    newsvd->maxprot, newsvd->type, newsvd->cred, NULL);
1729        }
1730 out:
1731        if (error == 0 && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1732                ASSERT(newsvd->amp == NULL);
1733                ASSERT(newsvd->tr_state == SEGVN_TR_OFF);
1734                newsvd->rcookie = svd->rcookie;
1735                hat_dup_region(newseg->s_as->a_hat, newsvd->rcookie);
1736        }
1737        return (error);
1738 }


1741 /*
1742  * callback function to invoke free_vp_pages() for only those pages actually
1743  * processed by the HAT when a shared region is destroyed.
1744  */
1745 extern int free_pages;

1747 static void
1748 segvn_hat_rgn_unload_callback(caddr_t saddr, caddr_t eaddr, caddr_t r_saddr,
1749     size_t r_size, void *r_obj, u_offset_t r_objoff)
1750 {
1751        u_offset_t off;
1752        size_t len;
1753        vnode_t *vp = (vnode_t *)r_obj;

1755        ASSERT(eaddr > saddr);
1756        ASSERT(saddr >= r_saddr);
1757        ASSERT(saddr < r_saddr + r_size);
1758        ASSERT(eaddr > r_saddr);
1759        ASSERT(eaddr <= r_saddr + r_size);
1760        ASSERT(vp != NULL);

1762        if (!free_pages) {
1763                return;
1764        }

1766        len = eaddr - saddr;
1767        off = (saddr - r_saddr) + r_objoff;
1768        free_vp_pages(vp, off, len);
1769 }

1771 /*
1772  * callback function used by segvn_unmap to invoke free_vp_pages() for only
1773  * those pages actually processed by the HAT
1774  */
1775 static void
1776 segvn_hat_unload_callback(hat_callback_t *cb)
```



```
1777 {
1778        struct seg              *seg = cb->hcb_data;
1779        struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
1780        size_t                  len;
1781        u_offset_t              off;

1783        ASSERT(svd->vp != NULL);
1784        ASSERT(cb->hcb_end_addr > cb->hcb_start_addr);
1785        ASSERT(cb->hcb_start_addr >= seg->s_base);

1787        len = cb->hcb_end_addr - cb->hcb_start_addr;
1788        off = cb->hcb_start_addr - seg->s_base;
1789        free_vp_pages(svd->vp, svd->offset + off, len);
1790 }

1792 /*
1793  * This function determines the number of bytes of swap reserved by
1794  * a segment for which per-page accounting is present. It is used to
1795  * calculate the correct value of a segvn_data's swresv.
1796  */
1797 static size_t
1798 segvn_count_swap_by_vpages(struct seg *seg)
1799 {
1800        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1801        struct vpage *vp, *evp;
1802        size_t nswappages = 0;

1804        ASSERT(svd->pageswap);
1805        ASSERT(svd->vpage != NULL);

1807        evp = &svd->vpage[seg_page(seg, seg->s_base + seg->s_size)];

1809        for (vp = svd->vpage; vp < evp; vp++) {
1810                if (VPP_ISSWAPRES(vp))
1811                        nswappages++;
1812        }

1814        return (nswappages << PAGESHIFT);
1815 }

1817 static int
1818 segvn_unmap(struct seg *seg, caddr_t addr, size_t len)
1819 {
1820        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1821        struct segvn_data *nsvd;
1822        struct seg *nseg;
1823        struct anon_map *amp;
1824        pgcnt_t opages;         /* old segment size in pages */
1825        pgcnt_t npages;         /* new segment size in pages */
1826        pgcnt_t dpages;         /* pages being deleted (unmapped) */
1827        hat_callback_t callback;        /* used for free_vp_pages() */
1828        hat_callback_t *cbp = NULL;
1829        caddr_t nbase;
1830        size_t nsize;
1831        size_t oswresv;
1832        int reclaim = 1;

1834        /*
1835         * We don't need any segment level locks for "segvn" data
1836         * since the address space is "write" locked.
1837         */
1838        ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1840        /*
1841         * Fail the unmap if pages are SOFTLOCKed through this mapping.
1842         * softlockcnt is protected from change by the as write lock.
```

```
1843          */
1844 retry:
1845         if (svd->softlockcnt > 0) {
1846                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1847
1848                 /*
1849                  * If this is shared segment non 0 softlockcnt
1850                  * means locked pages are still in use.
1851                  */
1852                 if (svd->type == MAP_SHARED) {
1853                         return (EAGAIN);
1854                 }
1855
1856                 /*
1857                  * since we do have the writers lock nobody can fill
1858                  * the cache during the purge. The flush either succeeds
1859                  * or we still have pending I/Os.
1860                  */
1861                 if (reclaim == 1) {
1862                         segvn_purge(seg);
1863                         reclaim = 0;
1864                         goto retry;
1865                 }
1866                 return (EAGAIN);
1867         }
1868
1869         /*
1870          * Check for bad sizes
1871          */
1872         if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
1873             (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET)) {
1874                 panic("segvn_unmap");
1875                 /*NOTREACHED*/
1876         }
1877
1878         if (seg->s_szc != 0) {
1879                 size_t pgsz = page_get_pagesize(seg->s_szc);
1880                 int err;
1881                 if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
1882                         ASSERT(seg->s_base != addr || seg->s_size != len);
1883                         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1884                                 ASSERT(svd->amp == NULL);
1885                                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1886                                 hat_leave_region(seg->s_as->a_hat,
1887                                     svd->rcookie, HAT_REGION_TEXT);
1888                                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
1889                                 /*
1890                                  * could pass a flag to segvn_demote_range()
1891                                  * below to tell it not to do any unloads but
1892                                  * this case is rare enough to not bother for
1893                                  * now.
1894                                  */
1895                         } else if (svd->tr_state == SEGVN_TR_INIT) {
1896                                 svd->tr_state = SEGVN_TR_OFF;
1897                         } else if (svd->tr_state == SEGVN_TR_ON) {
1898                                 ASSERT(svd->amp != NULL);
1899                                 segvn_textunrepl(seg, 1);
1900                                 ASSERT(svd->amp == NULL);
1901                                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1902                         }
1903                         VM_STAT_ADD(segvnvmstats.demoterange[0]);
1904                         err = segvn_demote_range(seg, addr, len, SDR_END, 0);
1905                         if (err == 0) {
1906                                 return (IE_RETRY);
1907                         }
1908                         return (err);
```



```
1909                 }
1910         }
1911
1912         /* Inform the vnode of the unmapping. */
1913         if (svd->vp) {
1914                 int error;
1915
1916                 error = VOP_DELMAP(svd->vp,
1917                     (offset_t)svd->offset + (uintptr_t)(addr - seg->s_base),
1918                     seg->s_as, addr, len, svd->prot, svd->maxprot,
1919                     svd->type, svd->cred, NULL);
1920
1921                 if (error == EAGAIN)
1922                         return (error);
1923         }
1924
1925         /*
1926          * Remove any page locks set through this mapping.
1927          * If text replication is not off no page locks could have been
1928          * established via this mapping.
1929          */
1930         if (svd->tr_state == SEGVN_TR_OFF) {
1931                 (void) segvn_lockop(seg, addr, len, 0, MC_UNLOCK, NULL, 0);
1932         }
1933
1934         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1935                 ASSERT(svd->amp == NULL);
1936                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1937                 ASSERT(svd->type == MAP_PRIVATE);
1938                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
1939                     HAT_REGION_TEXT);
1940                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
1941         } else if (svd->tr_state == SEGVN_TR_ON) {
1942                 ASSERT(svd->amp != NULL);
1943                 ASSERT(svd->pageprot == 0 && !(svd->prot & PROT_WRITE));
1944                 segvn_textunrepl(seg, 1);
1945                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
1946         } else {
1947                 if (svd->tr_state != SEGVN_TR_OFF) {
1948                         ASSERT(svd->tr_state == SEGVN_TR_INIT);
1949                         svd->tr_state = SEGVN_TR_OFF;
1950                 }
1951                 /*
1952                  * Unload any hardware translations in the range to be taken
1953                  * out. Use a callback to invoke free_vp_pages() effectively.
1954                  */
1955                 if (svd->vp != NULL && free_pages != 0) {
1956                         callback.hcb_data = seg;
1957                         callback.hcb_function = segvn_hat_unload_callback;
1958                         cbp = &callback;
1959                 }
1960                 hat_unload_callback(seg->s_as->a_hat, addr, len,
1961                     HAT_UNLOAD_UNMAP, cbp);
1962
1963                 if (svd->type == MAP_SHARED && svd->vp != NULL &&
1964                     (svd->vp->v_flag & VVMEXEC) &&
1965                     ((svd->prot & PROT_WRITE) || svd->pageprot)) {
1966                         segvn_inval_trcache(svd->vp);
1967                 }
1968         }
1969
1970         /*
1971          * Check for entire segment
1972          */
1973         if (addr == seg->s_base && len == seg->s_size) {
1974                 seg_free(seg);
```

```
1975                return (0);
1976        }

1978        opages = seg_pages(seg);
1979        dpages = btop(len);
1980        npages = opages - dpages;
1981        amp = svd->amp;
1982        ASSERT(amp == NULL || amp->a_szc >= seg->s_szc);

1984        /*
1985         * Check for beginning of segment
1986         */
1987        if (addr == seg->s_base) {
1988                if (svd->vpage != NULL) {
1989                        size_t nbytes;
1990                        struct vpage *ovpage;

1992                        ovpage = svd->vpage;    /* keep pointer to vpage */

1994                        nbytes = vpgtob(npages);
1995                        svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
1996                        bcopy(&ovpage[dpages], svd->vpage, nbytes);

1998                        /* free up old vpage */
1999                        kmem_free(ovpage, vpgtob(opages));
2000                }
2001                if (amp != NULL) {
2002                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2003                        if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2004                                /*
2005                                 * Shared anon map is no longer in use. Before
2006                                 * freeing its pages purge all entries from
2007                                 * pcache that belong to this amp.
2008                                 */
2009                                if (svd->type == MAP_SHARED) {
2010                                        ASSERT(amp->refcnt == 1);
2011                                        ASSERT(svd->softlockcnt == 0);
2012                                        anonmap_purge(amp);
2013                                }
2014                                /*
2015                                 * Free up now unused parts of anon_map array.
2016                                 */
2017                                if (amp->a_szc == seg->s_szc) {
2018                                        if (seg->s_szc != 0) {
2019                                                anon_free_pages(amp->ahp,
2020                                                    svd->anon_index, len,
2021                                                    seg->s_szc);
2022                                        } else {
2023                                                anon_free(amp->ahp,
2024                                                    svd->anon_index,
2025                                                    len);
2026                                        }
2027                                } else {
2028                                        ASSERT(svd->type == MAP_SHARED);
2029                                        ASSERT(amp->a_szc > seg->s_szc);
2030                                        anon_shmap_free_pages(amp,
2031                                            svd->anon_index, len);
2032                                }

2034                                /*
2035                                 * Unreserve swap space for the
2036                                 * unmapped chunk of this segment in
2037                                 * case it's MAP_SHARED
2038                                 */
2039                                if (svd->type == MAP_SHARED) {
2040                                        anon_unresv_zone(len,
```



```
2041                                            seg->s_as->a_proc->p_zone);
2042                                        amp->swresv -= len;
2043                                }
2044                        }
2045                        ANON_LOCK_EXIT(&amp->a_rwlock);
2046                        svd->anon_index += dpages;
2047                }
2048                if (svd->vp != NULL)
2049                        svd->offset += len;

2051                seg->s_base += len;
2052                seg->s_size -= len;

2054                if (svd->swresv) {
2055                        if (svd->flags & MAP_NORESERVE) {
2056                                ASSERT(amp);
2057                                oswresv = svd->swresv;

2059                                svd->swresv = ptob(anon_pages(amp->ahp,
2060                                    svd->anon_index, npages));
2061                                anon_unresv_zone(oswresv - svd->swresv,
2062                                    seg->s_as->a_proc->p_zone);
2063                                if (SEG_IS_PARTIAL_RESV(seg))
2064                                        seg->s_as->a_resvsize -= oswresv -
2065                                            svd->swresv;
2066                        } else {
2067                                size_t unlen;

2069                                if (svd->pageswap) {
2070                                        oswresv = svd->swresv;
2071                                        svd->swresv =
2072                                            segvn_count_swap_by_vpages(seg);
2073                                        ASSERT(oswresv >= svd->swresv);
2074                                        unlen = oswresv - svd->swresv;
2075                                } else {
2076                                        svd->swresv -= len;
2077                                        ASSERT(svd->swresv == seg->s_size);
2078                                        unlen = len;
2079                                }
2080                                anon_unresv_zone(unlen,
2081                                    seg->s_as->a_proc->p_zone);
2082                        }
2083                        TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2084                            seg, len, 0);
2085                }

2087                return (0);
2088        }

2090        /*
2091         * Check for end of segment
2092         */
2093        if (addr + len == seg->s_base + seg->s_size) {
2094                if (svd->vpage != NULL) {
2095                        size_t nbytes;
2096                        struct vpage *ovpage;

2098                        ovpage = svd->vpage;    /* keep pointer to vpage */

2100                        nbytes = vpgtob(npages);
2101                        svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
2102                        bcopy(ovpage, svd->vpage, nbytes);

2104                        /* free up old vpage */
2105                        kmem_free(ovpage, vpgtob(opages));
```

```
2107                }
2108                if (amp != NULL) {
2109                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2110                        if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2111                                /*
2112                                 * Free up now unused parts of anon_map array.
2113                                 */
2114                                ulong_t an_idx = svd->anon_index + npages;

2116                                /*
2117                                 * Shared anon map is no longer in use. Before
2118                                 * freeing its pages purge all entries from
2119                                 * pcache that belong to this amp.
2120                                 */
2121                                if (svd->type == MAP_SHARED) {
2122                                        ASSERT(amp->refcnt == 1);
2123                                        ASSERT(svd->softlockcnt == 0);
2124                                        anonmap_purge(amp);
2125                                }

2127                                if (amp->a_szc == seg->s_szc) {
2128                                        if (seg->s_szc != 0) {
2129                                                anon_free_pages(amp->ahp,
2130                                                    an_idx, len,
2131                                                    seg->s_szc);
2132                                        } else {
2133                                                anon_free(amp->ahp, an_idx,
2134                                                    len);
2135                                        }
2136                                } else {
2137                                        ASSERT(svd->type == MAP_SHARED);
2138                                        ASSERT(amp->a_szc > seg->s_szc);
2139                                        anon_shmap_free_pages(amp,
2140                                            an_idx, len);
2141                                }

2143                                /*
2144                                 * Unreserve swap space for the
2145                                 * unmapped chunk of this segment in
2146                                 * case it's MAP_SHARED
2147                                 */
2148                                if (svd->type == MAP_SHARED) {
2149                                        anon_unresv_zone(len,
2150                                            seg->s_as->a_proc->p_zone);
2151                                        amp->swresv -= len;
2152                                }
2153                        }
2154                        ANON_LOCK_EXIT(&amp->a_rwlock);
2155                }

2157                seg->s_size -= len;

2159                if (svd->swresv) {
2160                        if (svd->flags & MAP_NORESERVE) {
2161                                ASSERT(amp);
2162                                oswresv = svd->swresv;
2163                                svd->swresv = ptob(anon_pages(amp->ahp,
2164                                    svd->anon_index, npages));
2165                                anon_unresv_zone(oswresv - svd->swresv,
2166                                    seg->s_as->a_proc->p_zone);
2167                                if (SEG_IS_PARTIAL_RESV(seg))
2168                                        seg->s_as->a_resvsize -= oswresv -
2169                                            svd->swresv;
2170                        } else {
2171                                size_t unlen;
```



```
2173                                if (svd->pageswap) {
2174                                        oswresv = svd->swresv;
2175                                        svd->swresv =
2176                                            segvn_count_swap_by_vpages(seg);
2177                                        ASSERT(oswresv >= svd->swresv);
2178                                        unlen = oswresv - svd->swresv;
2179                                } else {
2180                                        svd->swresv -= len;
2181                                        ASSERT(svd->swresv == seg->s_size);
2182                                        unlen = len;
2183                                }
2184                                anon_unresv_zone(unlen,
2185                                    seg->s_as->a_proc->p_zone);
2186                        }
2187                        TRACE_3(TR_FAC_VM, TR_ANON_PROC,
2188                            "anon proc:%p %lu %u", seg, len, 0);
2189                }

2191                return (0);
2192        }

2194        /*
2195         * The section to go is in the middle of the segment,
2196         * have to make it into two segments.  nseg is made for
2197         * the high end while seg is cut down at the low end.
2198         */
2199        nbase = addr + len;                             /* new seg base */
2200        nsize = (seg->s_base + seg->s_size) - nbase;    /* new seg size */
2201        seg->s_size = addr - seg->s_base;               /* shrink old seg */
2202        nseg = seg_alloc(seg->s_as, nbase, nsize);
2203        if (nseg == NULL) {
2204                panic("segvn_unmap seg_alloc");
2205                /*NOTREACHED*/
2206        }
2207        nseg->s_ops = seg->s_ops;
2208        nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
2209        nseg->s_data = (void *)nsvd;
2210        nseg->s_szc = seg->s_szc;
2211        *nsvd = *svd;
2212        nsvd->seg = nseg;
2213        nsvd->offset = svd->offset + (uintptr_t)(nseg->s_base - seg->s_base);
2214        nsvd->swresv = 0;
2215        nsvd->softlockcnt = 0;
2216        nsvd->softlockcnt_sbase = 0;
2217        nsvd->softlockcnt_send = 0;
2218        ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);

2220        if (svd->vp != NULL) {
2221                VN_HOLD(nsvd->vp);
2222                if (nsvd->type == MAP_SHARED)
2223                        lgrp_shm_policy_init(NULL, nsvd->vp);
2224        }
2225        crhold(svd->cred);

2227        if (svd->vpage == NULL) {
2228                nsvd->vpage = NULL;
2229        } else {
2230                /* need to split vpage into two arrays */
2231                size_t nbytes;
2232                struct vpage *ovpage;

2234                ovpage = svd->vpage;            /* keep pointer to vpage */

2236                npages = seg_pages(seg);        /* seg has shrunk */
2237                nbytes = vpgtob(npages);
2238                svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
```

```
2240                bcopy(ovpage, svd->vpage, nbytes);

2242                npages = seg_pages(nseg);
2243                nbytes = vpgtob(npages);
2244                nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);

2246                bcopy(&ovpage[opages - npages], nsvd->vpage, nbytes);

2248                /* free up old vpage */
2249                kmem_free(ovpage, vpgtob(opages));
2250        }

2252        if (amp == NULL) {
2253                nsvd->amp = NULL;
2254                nsvd->anon_index = 0;
2255        } else {
2256                /*
2257                 * Need to create a new anon map for the new segment.
2258                 * We'll also allocate a new smaller array for the old
2259                 * smaller segment to save space.
2260                 */
2261                opages = btop((uintptr_t)(addr - seg->s_base));
2262                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2263                if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2264                        /*
2265                         * Free up now unused parts of anon_map array.
2266                         */
2267                        ulong_t an_idx = svd->anon_index + opages;

2269                        /*
2270                         * Shared anon map is no longer in use. Before
2271                         * freeing its pages purge all entries from
2272                         * pcache that belong to this amp.
2273                         */
2274                        if (svd->type == MAP_SHARED) {
2275                                ASSERT(amp->refcnt == 1);
2276                                ASSERT(svd->softlockcnt == 0);
2277                                anonmap_purge(amp);
2278                        }

2280                        if (amp->a_szc == seg->s_szc) {
2281                                if (seg->s_szc != 0) {
2282                                        anon_free_pages(amp->ahp, an_idx, len,
2283                                            seg->s_szc);
2284                                } else {
2285                                        anon_free(amp->ahp, an_idx,
2286                                            len);
2287                                }
2288                        } else {
2289                                ASSERT(svd->type == MAP_SHARED);
2290                                ASSERT(amp->a_szc > seg->s_szc);
2291                                anon_shmap_free_pages(amp, an_idx, len);
2292                        }

2294                        /*
2295                         * Unreserve swap space for the
2296                         * unmapped chunk of this segment in
2297                         * case it's MAP_SHARED
2298                         */
2299                        if (svd->type == MAP_SHARED) {
2300                                anon_unresv_zone(len,
2301                                    seg->s_as->a_proc->p_zone);
2302                                amp->swresv -= len;
2303                        }
2304                }
```



```
2305                nsvd->anon_index = svd->anon_index +
2306                    btop((uintptr_t)(nseg->s_base - seg->s_base));
2307                if (svd->type == MAP_SHARED) {
2308                        amp->refcnt++;
2309                        nsvd->amp = amp;
2310                } else {
2311                        struct anon_map *namp;
2312                        struct anon_hdr *nahp;

2314                        ASSERT(svd->type == MAP_PRIVATE);
2315                        nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
2316                        namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
2317                        namp->a_szc = seg->s_szc;
2318                        (void) anon_copy_ptr(amp->ahp, svd->anon_index, nahp,
2319                            0, btop(seg->s_size), ANON_SLEEP);
2320                        (void) anon_copy_ptr(amp->ahp, nsvd->anon_index,
2321                            namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
2322                        anon_release(amp->ahp, btop(amp->size));
2323                        svd->anon_index = 0;
2324                        nsvd->anon_index = 0;
2325                        amp->ahp = nahp;
2326                        amp->size = seg->s_size;
2327                        nsvd->amp = namp;
2328                }
2329                ANON_LOCK_EXIT(&amp->a_rwlock);
2330        }
2331        if (svd->swresv) {
2332                if (svd->flags & MAP_NORESERVE) {
2333                        ASSERT(amp);
2334                        oswresv = svd->swresv;
2335                        svd->swresv = ptob(anon_pages(amp->ahp,
2336                            svd->anon_index, btop(seg->s_size)));
2337                        nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
2338                            nsvd->anon_index, btop(nseg->s_size)));
2339                        ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
2340                        anon_unresv_zone(oswresv - (svd->swresv + nsvd->swresv),
2341                            seg->s_as->a_proc->p_zone);
2342                        if (SEG_IS_PARTIAL_RESV(seg))
2343                                seg->s_as->a_resvsize -= oswresv -
2344                                    (svd->swresv + nsvd->swresv);
2345                } else {
2346                        size_t unlen;

2348                        if (svd->pageswap) {
2349                                oswresv = svd->swresv;
2350                                svd->swresv = segvn_count_swap_by_vpages(seg);
2351                                nsvd->swresv = segvn_count_swap_by_vpages(nseg);
2352                                ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
2353                                unlen = oswresv - (svd->swresv + nsvd->swresv);
2354                        } else {
2355                                if (seg->s_size + nseg->s_size + len !=
2356                                    svd->swresv) {
2357                                        panic("segvn_unmap: cannot split "
2358                                            "swap reservation");
2359                                        /*NOTREACHED*/
2360                                }
2361                                svd->swresv = seg->s_size;
2362                                nsvd->swresv = nseg->s_size;
2363                                unlen = len;
2364                        }
2365                        anon_unresv_zone(unlen,
2366                            seg->s_as->a_proc->p_zone);
2367                }
2368                TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2369                    seg, len, 0);
2370        }
```

```
2372         return (0);                    /* I'm glad that's all over with! */
2373 }
2375 static void
2376 segvn_free(struct seg *seg)
2377 {
2378         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2379         pgcnt_t npages = seg_pages(seg);
2380         struct anon_map *amp;
2381         size_t len;
2383         /*
2384          * We don't need any segment level locks for "segvn" data
2385          * since the address space is "write" locked.
2386          */
2387         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
2388         ASSERT(svd->tr_state == SEGVN_TR_OFF);
2390         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
2392         /*
2393          * Be sure to unlock pages. XXX Why do things get free'ed instead
2394          * of unmapped? XXX
2395          */
2396         (void) segvn_lockop(seg, seg->s_base, seg->s_size,
2397             0, MC_UNLOCK, NULL, 0);
2399         /*
2400          * Deallocate the vpage and anon pointers if necessary and possible.
2401          */
2402         if (svd->vpage != NULL) {
2403                 kmem_free(svd->vpage, vpgtob(npages));
2404                 svd->vpage = NULL;
2405         }
2406         if ((amp = svd->amp) != NULL) {
2407                 /*
2408                  * If there are no more references to this anon_map
2409                  * structure, then deallocate the structure after freeing
2410                  * up all the anon slot pointers that we can.
2411                  */
2412                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2413                 ASSERT(amp->a_szc >= seg->s_szc);
2414                 if (--amp->refcnt == 0) {
2415                         if (svd->type == MAP_PRIVATE) {
2416                                 /*
2417                                  * Private - we only need to anon_free
2418                                  * the part that this segment refers to.
2419                                  */
2420                                 if (seg->s_szc != 0) {
2421                                         anon_free_pages(amp->ahp,
2422                                             svd->anon_index, seg->s_size,
2423                                             seg->s_szc);
2424                                 } else {
2425                                         anon_free(amp->ahp, svd->anon_index,
2426                                             seg->s_size);
2427                                 }
2428                         } else {
2430                                 /*
2431                                  * Shared anon map is no longer in use. Before
2432                                  * freeing its pages purge all entries from
2433                                  * pcache that belong to this amp.
2434                                  */
2435                                 ASSERT(svd->softlockcnt == 0);
2436                                 anonmap_purge(amp);
```



```
2438                                 /*
2439                                  * Shared - anon_free the entire
2440                                  * anon_map's worth of stuff and
2441                                  * release any swap reservation.
2442                                  */
2443                                 if (amp->a_szc != 0) {
2444                                         anon_shmap_free_pages(amp, 0,
2445                                             amp->size);
2446                                 } else {
2447                                         anon_free(amp->ahp, 0, amp->size);
2448                                 }
2449                                 if ((len = amp->swresv) != 0) {
2450                                         anon_unresv_zone(len,
2451                                             seg->s_as->a_proc->p_zone);
2452                                         TRACE_3(TR_FAC_VM, TR_ANON_PROC,
2453                                             "anon proc:%p %lu %u", seg, len, 0);
2454                                 }
2455                         }
2456                         svd->amp = NULL;
2457                         ANON_LOCK_EXIT(&amp->a_rwlock);
2458                         anonmap_free(amp);
2459                 } else if (svd->type == MAP_PRIVATE) {
2460                         /*
2461                          * We had a private mapping which still has
2462                          * a held anon_map so just free up all the
2463                          * anon slot pointers that we were using.
2464                          */
2465                         if (seg->s_szc != 0) {
2466                                 anon_free_pages(amp->ahp, svd->anon_index,
2467                                     seg->s_size, seg->s_szc);
2468                         } else {
2469                                 anon_free(amp->ahp, svd->anon_index,
2470                                     seg->s_size);
2471                         }
2472                         ANON_LOCK_EXIT(&amp->a_rwlock);
2473                 } else {
2474                         ANON_LOCK_EXIT(&amp->a_rwlock);
2475                 }
2476         }
2478         /*
2479          * Release swap reservation.
2480          */
2481         if ((len = svd->swresv) != 0) {
2482                 anon_unresv_zone(svd->swresv,
2483                     seg->s_as->a_proc->p_zone);
2484                 TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
2485                     seg, len, 0);
2486                 if (SEG_IS_PARTIAL_RESV(seg))
2487                         seg->s_as->a_resvsize -= svd->swresv;
2488                 svd->swresv = 0;
2489         }
2490         /*
2491          * Release claim on vnode, credentials, and finally free the
2492          * private data.
2493          */
2494         if (svd->vp != NULL) {
2495                 if (svd->type == MAP_SHARED)
2496                         lgrp_shm_policy_fini(NULL, svd->vp);
2497                 VN_RELE(svd->vp);
2498                 svd->vp = NULL;
2499         }
2500         crfree(svd->cred);
2501         svd->pageprot = 0;
2502         svd->pageadvice = 0;
```

```
2503         svd->pageswap = 0;
2504         svd->cred = NULL;

2506         /*
2507          * Take segfree_syncmtx lock to let segvn_reclaim() finish if it's
2508          * still working with this segment without holding as lock (in case
2509          * it's called by pcache async thread).
2510          */
2511         ASSERT(svd->softlockcnt == 0);
2512         mutex_enter(&svd->segfree_syncmtx);
2513         mutex_exit(&svd->segfree_syncmtx);

2515         seg->s_data = NULL;
2516         kmem_cache_free(segvn_cache, svd);
2517 }

2519 /*
2520  * Do a F_SOFTUNLOCK call over the range requested.  The range must have
2521  * already been F_SOFTLOCK'ed.
2522  * Caller must always match addr and len of a softunlock with a previous
2523  * softlock with exactly the same addr and len.
2524  */
2525 static void
2526 segvn_softunlock(struct seg *seg, caddr_t addr, size_t len, enum seg_rw rw)
2527 {
2528         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2529         page_t *pp;
2530         caddr_t adr;
2531         struct vnode *vp;
2532         u_offset_t offset;
2533         ulong_t anon_index;
2534         struct anon_map *amp;
2535         struct anon *ap = NULL;

2537         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
2538         ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

2540         if ((amp = svd->amp) != NULL)
2541                 anon_index = svd->anon_index + seg_page(seg, addr);

2543         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
2544                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
2545                 hat_unlock_region(seg->s_as->a_hat, addr, len, svd->rcookie);
2546         } else {
2547                 hat_unlock(seg->s_as->a_hat, addr, len);
2548         }
2549         for (adr = addr; adr < addr + len; adr += PAGESIZE) {
2550                 if (amp != NULL) {
2551                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2552                         if ((ap = anon_get_ptr(amp->ahp, anon_index++))
2553                             != NULL) {
2554                                 swap_xlate(ap, &vp, &offset);
2555                         } else {
2556                                 vp = svd->vp;
2557                                 offset = svd->offset +
2558                                     (uintptr_t)(adr - seg->s_base);
2559                         }
2560                         ANON_LOCK_EXIT(&amp->a_rwlock);
2561                 } else {
2562                         vp = svd->vp;
2563                         offset = svd->offset +
2564                             (uintptr_t)(adr - seg->s_base);
2565                 }

2567                 /*
2568                  * Use page_find() instead of page_lookup() to
```



```
2569                  * find the page since we know that it is locked.
2570                  */
2571                 pp = page_find(vp, offset);
2572                 if (pp == NULL) {
2573                         panic(
2574                             "segvn_softunlock: addr %p, ap %p, vp %p, off %llx",
2575                             (void *)adr, (void *)ap, (void *)vp, offset);
2576                         /*NOTREACHED*/
2577                 }

2579                 if (rw == S_WRITE) {
2580                         hat_setrefmod(pp);
2581                         if (seg->s_as->a_vbits)
2582                                 hat_setstat(seg->s_as, adr, PAGESIZE,
2583                                     P_REF | P_MOD);
2584                 } else if (rw != S_OTHER) {
2585                         hat_setref(pp);
2586                         if (seg->s_as->a_vbits)
2587                                 hat_setstat(seg->s_as, adr, PAGESIZE, P_REF);
2588                 }
2589                 TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
2590                     "segvn_fault:pp %p vp %p offset %llx", pp, vp, offset);
2591                 page_unlock(pp);
2592         }
2593         ASSERT(svd->softlockcnt >= btop(len));
2594         if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -btop(len))) {
2595                 /*
2596                  * All SOFTLOCKS are gone. Wakeup any waiting
2597                  * unmappers so they can try again to unmap.
2598                  * Check for waiters first without the mutex
2599                  * held so we don't always grab the mutex on
2600                  * softunlocks.
2601                  */
2602                 if (AS_ISUNMAPWAIT(seg->s_as)) {
2603                         mutex_enter(&seg->s_as->a_contents);
2604                         if (AS_ISUNMAPWAIT(seg->s_as)) {
2605                                 AS_CLRUNMAPWAIT(seg->s_as);
2606                                 cv_broadcast(&seg->s_as->a_cv);
2607                         }
2608                         mutex_exit(&seg->s_as->a_contents);
2609                 }
2610         }
2611 }

2613 #define PAGE_HANDLED    ((page_t *)-1)

2615 /*
2616  * Release all the pages in the NULL terminated ppp list
2617  * which haven't already been converted to PAGE_HANDLED.
2618  */
2619 static void
2620 segvn_pagelist_rele(page_t **ppp)
2621 {
2622         for (; *ppp != NULL; ppp++) {
2623                 if (*ppp != PAGE_HANDLED)
2624                         page_unlock(*ppp);
2625         }
2626 }

2628 static int stealcow = 1;

2630 /*
2631  * Workaround for viking chip bug.  See bug id 1220902.
2632  * To fix this down in pagefault() would require importing so
2633  * much as and segvn code as to be unmaintainable.
2634  */
```

```
2635 int enable_mbit_wa = 0;

2637 /*
2638  * Handles all the dirty work of getting the right
2639  * anonymous pages and loading up the translations.
2640  * This routine is called only from segvn_fault()
2641  * when looping over the range of addresses requested.
2642  *
2643  * The basic algorithm here is:
2644  *      If this is an anon_zero case
2645  *              Call anon_zero to allocate page
2646  *              Load up translation
2647  *              Return
2648  *      endif
2649  *      If this is an anon page
2650  *              Use anon_getpage to get the page
2651  *      else
2652  *              Find page in pl[] list passed in
2653  *      endif
2654  *      If not a cow
2655  *              Load up the translation to the page
2656  *              return
2657  *      endif
2658  *      Call anon_private to handle cow
2659  *      Load up (writable) translation to new page
2660  */
2661 static faultcode_t
2662 segvn_faultpage(
2663         struct hat *hat,                /* the hat to use for mapping */
2664         struct seg *seg,                /* seg_vn of interest */
2665         caddr_t addr,                   /* address in as */
2666         u_offset_t off,                 /* offset in vp */
2667         struct vpage *vpage,            /* pointer to vpage for vp, off */
2668         page_t *pl[],                   /* object source page pointer */
2669         uint_t vpprot,                  /* access allowed to object pages */
2670         enum fault_type type,           /* type of fault */
2671         enum seg_rw rw,                 /* type of access at fault */
2672         int brkcow)                     /* we may need to break cow */
2673 {
2674         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2675         page_t *pp, **ppp;
2676         uint_t pageflags = 0;
2677         page_t *anon_pl[1 + 1];
2678         page_t *opp = NULL;             /* original page */
2679         uint_t prot;
2680         int err;
2681         int cow;
2682         int claim;
2683         int steal = 0;
2684         ulong_t anon_index;
2685         struct anon *ap, *oldap;
2686         struct anon_map *amp;
2687         int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
2688         int anon_lock = 0;
2689         anon_sync_obj_t cookie;

2691         if (svd->flags & MAP_TEXT) {
2692                 hat_flag |= HAT_LOAD_TEXT;
2693         }

2695         ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
2696         ASSERT(seg->s_szc == 0);
2697         ASSERT(svd->tr_state != SEGVN_TR_INIT);

2699         /*
2700          * Initialize protection value for this page.
```



```
2701          * If we have per page protection values check it now.
2702          */
2703         if (svd->pageprot) {
2704                 uint_t protchk;

2706                 switch (rw) {
2707                 case S_READ:
2708                         protchk = PROT_READ;
2709                         break;
2710                 case S_WRITE:
2711                         protchk = PROT_WRITE;
2712                         break;
2713                 case S_EXEC:
2714                         protchk = PROT_EXEC;
2715                         break;
2716                 case S_OTHER:
2717                 default:
2718                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
2719                         break;
2720                 }

2722                 prot = VPP_PROT(vpage);
2723                 if ((prot & protchk) == 0)
2724                         return (FC_PROT);       /* illegal access type */
2725         } else {
2726                 prot = svd->prot;
2727         }

2729         if (type == F_SOFTLOCK) {
2730                 atomic_inc_ulong((ulong_t *)&svd->softlockcnt);
2731         }

2733         /*
2734          * Always acquire the anon array lock to prevent 2 threads from
2735          * allocating separate anon slots for the same "addr".
2736          */

2738         if ((amp = svd->amp) != NULL) {
2739                 ASSERT(RW_READ_HELD(&amp->a_rwlock));
2740                 anon_index = svd->anon_index + seg_page(seg, addr);
2741                 anon_array_enter(amp, anon_index, &cookie);
2742                 anon_lock = 1;
2743         }

2745         if (svd->vp == NULL && amp != NULL) {
2746                 if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL) {
2747                         /*
2748                          * Allocate a (normally) writable anonymous page of
2749                          * zeroes. If no advance reservations, reserve now.
2750                          */
2751                         if (svd->flags & MAP_NORESERVE) {
2752                                 if (anon_resv_zone(ptob(1),
2753                                     seg->s_as->a_proc->p_zone)) {
2754                                         atomic_add_long(&svd->swresv, ptob(1));
2755                                         atomic_add_long(&seg->s_as->a_resvsize,
2756                                             ptob(1));
2757                                 } else {
2758                                         err = ENOMEM;
2759                                         goto out;
2760                                 }
2761                         }
2762                         if ((pp = anon_zero(seg, addr, &ap,
2763                             svd->cred)) == NULL) {
2764                                 err = ENOMEM;
2765                                 goto out;       /* out of swap space */
2766                         }
```

```
2767                        /*
2768                         * Re-acquire the anon_map lock and
2769                         * initialize the anon array entry.
2770                         */
2771                        (void) anon_set_ptr(amp->ahp, anon_index, ap,
2772                            ANON_SLEEP);

2774                        ASSERT(pp->p_szc == 0);

2776                        /*
2777                         * Handle pages that have been marked for migration
2778                         */
2779                        if (lgrp_optimizations())
2780                                page_migrate(seg, addr, &pp, 1);

2782                        if (enable_mbit_wa) {
2783                                if (rw == S_WRITE)
2784                                        hat_setmod(pp);
2785                                else if (!hat_ismod(pp))
2786                                        prot &= ~PROT_WRITE;
2787                        }
2788                        /*
2789                         * If AS_PAGLCK is set in a_flags (via memcntl(2)
2790                         * with MC_LOCKAS, MCL_FUTURE) and this is a
2791                         * MAP_NORESERVE segment, we may need to
2792                         * permanently lock the page as it is being faulted
2793                         * for the first time. The following text applies
2794                         * only to MAP_NORESERVE segments:
2795                         *
2796                         * As per memcntl(2), if this segment was created
2797                         * after MCL_FUTURE was applied (a "future"
2798                         * segment), its pages must be locked.  If this
2799                         * segment existed at MCL_FUTURE application (a
2800                         * "past" segment), the interface is unclear.
2801                         *
2802                         * We decide to lock only if vpage is present:
2803                         *
2804                         * - "future" segments will have a vpage array (see
2805                         *    as_map), and so will be locked as required
2806                         *
2807                         * - "past" segments may not have a vpage array,
2808                         *    depending on whether events (such as
2809                         *    mprotect) have occurred. Locking if vpage
2810                         *    exists will preserve legacy behavior.  Not
2811                         *    locking if vpage is absent, will not break
2812                         *    the interface or legacy behavior.  Note that
2813                         *    allocating vpage here if it's absent requires
2814                         *    upgrading the segvn reader lock, the cost of
2815                         *    which does not seem worthwhile.
2816                         *
2817                         * Usually testing and setting VPP_ISPPLOCK and
2818                         * VPP_SETPPLOCK requires holding the segvn lock as
2819                         * writer, but in this case all readers are
2820                         * serializing on the anon array lock.
2821                         */
2822                        if (AS_ISPGLCK(seg->s_as) && vpage != NULL &&
2823                            (svd->flags & MAP_NORESERVE) &&
2824                            !VPP_ISPPLOCK(vpage)) {
2825                                proc_t *p = seg->s_as->a_proc;
2826                                ASSERT(svd->type == MAP_PRIVATE);
2827                                mutex_enter(&p->p_lock);
2828                                if (rctl_incr_locked_mem(p, NULL, PAGESIZE,
2829                                    1) == 0) {
2830                                        claim = VPP_PROT(vpage) & PROT_WRITE;
2831                                        if (page_pp_lock(pp, claim, 0)) {
2832                                                VPP_SETPPLOCK(vpage);
```



```
2833                                        } else {
2834                                                rctl_decr_locked_mem(p, NULL,
2835                                                    PAGESIZE, 1);
2836                                        }
2837                                }
2838                                mutex_exit(&p->p_lock);
2839                        }

2841                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
2842                        hat_memload(hat, addr, pp, prot, hat_flag);

2844                        if (!(hat_flag & HAT_LOAD_LOCK))
2845                                page_unlock(pp);

2847                        anon_array_exit(&cookie);
2848                        return (0);
2849                }
2850        }

2852        /*
2853         * Obtain the page structure via anon_getpage() if it is
2854         * a private copy of an object (the result of a previous
2855         * copy-on-write).
2856         */
2857        if (amp != NULL) {
2858                if ((ap = anon_get_ptr(amp->ahp, anon_index)) != NULL) {
2859                        err = anon_getpage(&ap, &vpprot, anon_pl, PAGESIZE,
2860                            seg, addr, rw, svd->cred);
2861                        if (err)
2862                                goto out;

2864                        if (svd->type == MAP_SHARED) {
2865                                /*
2866                                 * If this is a shared mapping to an
2867                                 * anon_map, then ignore the write
2868                                 * permissions returned by anon_getpage().
2869                                 * They apply to the private mappings
2870                                 * of this anon_map.
2871                                 */
2872                                vpprot |= PROT_WRITE;
2873                        }
2874                        opp = anon_pl[0];
2875                }
2876        }

2878        /*
2879         * Search the pl[] list passed in if it is from the
2880         * original object (i.e., not a private copy).
2881         */
2882        if (opp == NULL) {
2883                /*
2884                 * Find original page.  We must be bringing it in
2885                 * from the list in pl[].
2886                 */
2887                for (ppp = pl; (opp = *ppp) != NULL; ppp++) {
2888                        if (opp == PAGE_HANDLED)
2889                                continue;
2890                        ASSERT(opp->p_vnode == svd->vp); /* XXX */
2891                        if (opp->p_offset == off)
2892                                break;
2893                }
2894                if (opp == NULL) {
2895                        panic("segvn_faultpage not found");
2896                        /*NOTREACHED*/
2897                }
2898                *ppp = PAGE_HANDLED;
```

```
2900         }

2902         ASSERT(PAGE_LOCKED(opp));

2904         TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
2905             "segvn_fault:pp %p vp %p offset %llx", opp, NULL, 0);

2907         /*
2908          * The fault is treated as a copy-on-write fault if a
2909          * write occurs on a private segment and the object
2910          * page (i.e., mapping) is write protected.  We assume
2911          * that fatal protection checks have already been made.
2912          */

2914         if (brkcow) {
2915                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
2916                 cow = !(vpprot & PROT_WRITE);
2917         } else if (svd->tr_state == SEGVN_TR_ON) {
2918                 /*
2919                  * If we are doing text replication COW on first touch.
2920                  */
2921                 ASSERT(amp != NULL);
2922                 ASSERT(svd->vp != NULL);
2923                 ASSERT(rw != S_WRITE);
2924                 cow = (ap == NULL);
2925         } else {
2926                 cow = 0;
2927         }

2929         /*
2930          * If not a copy-on-write case load the translation
2931          * and return.
2932          */
2933         if (cow == 0) {

2935                 /*
2936                  * Handle pages that have been marked for migration
2937                  */
2938                 if (lgrp_optimizations())
2939                         page_migrate(seg, addr, &opp, 1);

2941                 if (IS_VMODSORT(opp->p_vnode) || enable_mbit_wa) {
2942                         if (rw == S_WRITE)
2943                                 hat_setmod(opp);
2944                         else if (rw != S_OTHER && !hat_ismod(opp))
2945                                 prot &= ~PROT_WRITE;
2946                 }

2948                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
2949                     (!svd->pageprot && svd->prot == (prot & vpprot)));
2950                 ASSERT(amp == NULL ||
2951                     svd->rcookie == HAT_INVALID_REGION_COOKIE);
2952                 hat_memload_region(hat, addr, opp, prot & vpprot, hat_flag,
2953                     svd->rcookie);

2955                 if (!(hat_flag & HAT_LOAD_LOCK))
2956                         page_unlock(opp);

2958                 if (anon_lock) {
2959                         anon_array_exit(&cookie);
2960                 }
2961                 return (0);
2962         }

2964         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
```



```
2966         hat_setref(opp);

2968         ASSERT(amp != NULL && anon_lock);

2970         /*
2971          * Steal the page only if it isn't a private page
2972          * since stealing a private page is not worth the effort.
2973          */
2974         if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL)
2975                 steal = 1;

2977         /*
2978          * Steal the original page if the following conditions are true:
2979          *
2980          * We are low on memory, the page is not private, page is not large,
2981          * not shared, not modified, not 'locked' or if we have it 'locked'
2982          * (i.e., p_cowcnt == 1 and p_lckcnt == 0, which also implies
2983          * that the page is not shared) and if it doesn't have any
2984          * translations. page_struct_lock isn't needed to look at p_cowcnt
2985          * and p_lckcnt because we first get exclusive lock on page.
2986          */
2987         (void) hat_pagesync(opp, HAT_SYNC_DONTZERO | HAT_SYNC_STOPON_MOD);

2989         if (stealcow && freemem < minfree && steal && opp->p_szc == 0 &&
2990             page_tryupgrade(opp) && !hat_ismod(opp) &&
2991             ((opp->p_lckcnt == 0 && opp->p_cowcnt == 0) ||
2992             (opp->p_lckcnt == 0 && opp->p_cowcnt == 1 &&
2993             vpage != NULL && VPP_ISPPLOCK(vpage)))) {
2994                 /*
2995                  * Check if this page has other translations
2996                  * after unloading our translation.
2997                  */
2998                 if (hat_page_is_mapped(opp)) {
2999                         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
3000                         hat_unload(seg->s_as->a_hat, addr, PAGESIZE,
3001                             HAT_UNLOAD);
3002                 }

3004                 /*
3005                  * hat_unload() might sync back someone else's recent
3006                  * modification, so check again.
3007                  */
3008                 if (!hat_ismod(opp) && !hat_page_is_mapped(opp))
3009                         pageflags |= STEAL_PAGE;
3010         }

3012         /*
3013          * If we have a vpage pointer, see if it indicates that we have
3014          * ''locked'' the page we map -- if so, tell anon_private to
3015          * transfer the locking resource to the new page.
3016          *
3017          * See Statement at the beginning of segvn_lockop regarding
3018          * the way lockcnts/cowcnts are handled during COW.
3019          *
3020          */
3021         if (vpage != NULL && VPP_ISPPLOCK(vpage))
3022                 pageflags |= LOCK_PAGE;

3024         /*
3025          * Allocate a private page and perform the copy.
3026          * For MAP_NORESERVE reserve swap space now, unless this
3027          * is a cow fault on an existing anon page in which case
3028          * MAP_NORESERVE will have made advance reservations.
3029          */
3030         if ((svd->flags & MAP_NORESERVE) && (ap == NULL)) {
```

```
3031                 if (anon_resv_zone(ptob(1), seg->s_as->a_proc->p_zone)) {
3032                         atomic_add_long(&svd->swresv, ptob(1));
3033                         atomic_add_long(&seg->s_as->a_resvsize, ptob(1));
3034                 } else {
3035                         page_unlock(opp);
3036                         err = ENOMEM;
3037                         goto out;
3038                 }
3039         }
3040         oldap = ap;
3041         pp = anon_private(&ap, seg, addr, prot, opp, pageflags, svd->cred);
3042         if (pp == NULL) {
3043                 err = ENOMEM;   /* out of swap space */
3044                 goto out;
3045         }

3047         /*
3048          * If we copied away from an anonymous page, then
3049          * we are one step closer to freeing up an anon slot.
3050          *
3051          * NOTE:  The original anon slot must be released while
3052          * holding the "anon_map" lock.  This is necessary to prevent
3053          * other threads from obtaining a pointer to the anon slot
3054          * which may be freed if its "refcnt" is 1.
3055          */
3056         if (oldap != NULL)
3057                 anon_decref(oldap);

3059         (void) anon_set_ptr(amp->ahp, anon_index, ap, ANON_SLEEP);

3061         /*
3062          * Handle pages that have been marked for migration
3063          */
3064         if (lgrp_optimizations())
3065                 page_migrate(seg, addr, &pp, 1);

3067         ASSERT(pp->p_szc == 0);

3069         ASSERT(!IS_VMODSORT(pp->p_vnode));
3070         if (enable_mbit_wa) {
3071                 if (rw == S_WRITE)
3072                         hat_setmod(pp);
3073                 else if (!hat_ismod(pp))
3074                         prot &= ~PROT_WRITE;
3075         }

3077         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
3078         hat_memload(hat, addr, pp, prot, hat_flag);

3080         if (!(hat_flag & HAT_LOAD_LOCK))
3081                 page_unlock(pp);

3083         ASSERT(anon_lock);
3084         anon_array_exit(&cookie);
3085         return (0);
3086 out:
3087         if (anon_lock)
3088                 anon_array_exit(&cookie);

3090         if (type == F_SOFTLOCK) {
3091                 atomic_dec_ulong((ulong_t *)&svd->softlockcnt);
3092         }
3093         return (FC_MAKE_ERR(err));
3094 }

3096 /*
```



```
3097  * relocate a bunch of smaller targ pages into one large repl page. all targ
3098  * pages must be complete pages smaller than replacement pages.
3099  * it's assumed that no page's szc can change since they are all PAGESIZE or
3100  * complete large pages locked SHARED.
3101  */
3102 static void
3103 segvn_relocate_pages(page_t **targ, page_t *replacement)
3104 {
3105         page_t *pp;
3106         pgcnt_t repl_npgs, curnpgs;
3107         pgcnt_t i;
3108         uint_t repl_szc = replacement->p_szc;
3109         page_t *first_repl = replacement;
3110         page_t *repl;
3111         spgcnt_t npgs;

3113         VM_STAT_ADD(segvnvmstats.relocatepages[0]);

3115         ASSERT(repl_szc != 0);
3116         npgs = repl_npgs = page_get_pagecnt(repl_szc);

3118         i = 0;
3119         while (repl_npgs) {
3120                 spgcnt_t nreloc;
3121                 int err;
3122                 ASSERT(replacement != NULL);
3123                 pp = targ[i];
3124                 ASSERT(pp->p_szc < repl_szc);
3125                 ASSERT(PAGE_EXCL(pp));
3126                 ASSERT(!PP_ISFREE(pp));
3127                 curnpgs = page_get_pagecnt(pp->p_szc);
3128                 if (curnpgs == 1) {
3129                         VM_STAT_ADD(segvnvmstats.relocatepages[1]);
3130                         repl = replacement;
3131                         page_sub(&replacement, repl);
3132                         ASSERT(PAGE_EXCL(repl));
3133                         ASSERT(!PP_ISFREE(repl));
3134                         ASSERT(repl->p_szc == repl_szc);
3135                 } else {
3136                         page_t *repl_savepp;
3137                         int j;
3138                         VM_STAT_ADD(segvnvmstats.relocatepages[2]);
3139                         repl_savepp = replacement;
3140                         for (j = 0; j < curnpgs; j++) {
3141                                 repl = replacement;
3142                                 page_sub(&replacement, repl);
3143                                 ASSERT(PAGE_EXCL(repl));
3144                                 ASSERT(!PP_ISFREE(repl));
3145                                 ASSERT(repl->p_szc == repl_szc);
3146                                 ASSERT(page_pptonum(targ[i + j]) ==
3147                                     page_pptonum(targ[i]) + j);
3148                         }
3149                         repl = repl_savepp;
3150                         ASSERT(IS_P2ALIGNED(page_pptonum(repl), curnpgs));
3151                 }
3152                 err = page_relocate(&pp, &repl, 0, 1, &nreloc, NULL);
3153                 if (err || nreloc != curnpgs) {
3154                         panic("segvn_relocate_pages: "
3155                             "page_relocate failed err=%d curnpgs=%ld "
3156                             "nreloc=%ld", err, curnpgs, nreloc);
3157                 }
3158                 ASSERT(curnpgs <= repl_npgs);
3159                 repl_npgs -= curnpgs;
3160                 i += curnpgs;
3161         }
3162         ASSERT(replacement == NULL);
```

```
3164         repl = first_repl;
3165         repl_npgs = npgs;
3166         for (i = 0; i < repl_npgs; i++) {
3167                 ASSERT(PAGE_EXCL(repl));
3168                 ASSERT(!PP_ISFREE(repl));
3169                 targ[i] = repl;
3170                 page_downgrade(targ[i]);
3171                 repl++;
3172         }
3173 }

3175 /*
3176  * Check if all pages in ppa array are complete smaller than szc pages and
3177  * their roots will still be aligned relative to their current size if the
3178  * entire ppa array is relocated into one szc page. If these conditions are
3179  * not met return 0.
3180  *
3181  * If all pages are properly aligned attempt to upgrade their locks
3182  * to exclusive mode. If it fails set *upgrdfail to 1 and return 0.
3183  * upgrdfail was set to 0 by caller.
3184  *
3185  * Return 1 if all pages are aligned and locked exclusively.
3186  *
3187  * If all pages in ppa array happen to be physically contiguous to make one
3188  * szc page and all exclusive locks are successfully obtained promote the page
3189  * size to szc and set *pszc to szc. Return 1 with pages locked shared.
3190  */
3191 static int
3192 segvn_full_szcpages(page_t **ppa, uint_t szc, int *upgrdfail, uint_t *pszc)
3193 {
3194         page_t *pp;
3195         pfn_t pfn;
3196         pgcnt_t totnpgs = page_get_pagecnt(szc);
3197         pfn_t first_pfn;
3198         int contig = 1;
3199         pgcnt_t i;
3200         pgcnt_t j;
3201         uint_t curszc;
3202         pgcnt_t curnpgs;
3203         int root = 0;

3205         ASSERT(szc > 0);

3207         VM_STAT_ADD(segvnvmstats.fullszcpages[0]);

3209         for (i = 0; i < totnpgs; i++) {
3210                 pp = ppa[i];
3211                 ASSERT(PAGE_SHARED(pp));
3212                 ASSERT(!PP_ISFREE(pp));
3213                 pfn = page_pptonum(pp);
3214                 if (i == 0) {
3215                         if (!IS_P2ALIGNED(pfn, totnpgs)) {
3216                                 contig = 0;
3217                         } else {
3218                                 first_pfn = pfn;
3219                         }
3220                 } else if (contig && pfn != first_pfn + i) {
3221                         contig = 0;
3222                 }
3223                 if (pp->p_szc == 0) {
3224                         if (root) {
3225                                 VM_STAT_ADD(segvnvmstats.fullszcpages[1]);
3226                                 return (0);
3227                         }
3228                 } else if (!root) {
```



```
3229                         if ((curszc = pp->p_szc) >= szc) {
3230                                 VM_STAT_ADD(segvnvmstats.fullszcpages[2]);
3231                                 return (0);
3232                         }
3233                         if (curszc == 0) {
3234                                 /*
3235                                  * p_szc changed means we don't have all pages
3236                                  * locked. return failure.
3237                                  */
3238                                 VM_STAT_ADD(segvnvmstats.fullszcpages[3]);
3239                                 return (0);
3240                         }
3241                         curnpgs = page_get_pagecnt(curszc);
3242                         if (!IS_P2ALIGNED(pfn, curnpgs) ||
3243                             !IS_P2ALIGNED(i, curnpgs)) {
3244                                 VM_STAT_ADD(segvnvmstats.fullszcpages[4]);
3245                                 return (0);
3246                         }
3247                         root = 1;
3248                 } else {
3249                         ASSERT(i > 0);
3250                         VM_STAT_ADD(segvnvmstats.fullszcpages[5]);
3251                         if (pp->p_szc != curszc) {
3252                                 VM_STAT_ADD(segvnvmstats.fullszcpages[6]);
3253                                 return (0);
3254                         }
3255                         if (pfn - 1 != page_pptonum(ppa[i - 1])) {
3256                                 panic("segvn_full_szcpages: "
3257                                     "large page not physically contiguous");
3258                         }
3259                         if (P2PHASE(pfn, curnpgs) == curnpgs - 1) {
3260                                 root = 0;
3261                         }
3262                 }
3263         }

3265         for (i = 0; i < totnpgs; i++) {
3266                 ASSERT(ppa[i]->p_szc < szc);
3267                 if (!page_tryupgrade(ppa[i])) {
3268                         for (j = 0; j < i; j++) {
3269                                 page_downgrade(ppa[j]);
3270                         }
3271                         *pszc = ppa[i]->p_szc;
3272                         *upgrdfail = 1;
3273                         VM_STAT_ADD(segvnvmstats.fullszcpages[7]);
3274                         return (0);
3275                 }
3276         }

3278         /*
3279          * When a page is put a free cachelist its szc is set to 0.  if file
3280          * system reclaimed pages from cachelist targ pages will be physically
3281          * contiguous with 0 p_szc.  in this case just upgrade szc of targ
3282          * pages without any relocations.
3283          * To avoid any hat issues with previous small mappings
3284          * hat_pageunload() the target pages first.
3285          */
3286         if (contig) {
3287                 VM_STAT_ADD(segvnvmstats.fullszcpages[8]);
3288                 for (i = 0; i < totnpgs; i++) {
3289                         (void) hat_pageunload(ppa[i], HAT_FORCE_PGUNLOAD);
3290                 }
3291                 for (i = 0; i < totnpgs; i++) {
3292                         ppa[i]->p_szc = szc;
3293                 }
3294                 for (i = 0; i < totnpgs; i++) {
```

```
3295                         ASSERT(PAGE_EXCL(ppa[i]));
3296                         page_downgrade(ppa[i]);
3297                 }
3298                 if (pszc != NULL) {
3299                         *pszc = szc;
3300                 }
3301         }
3302         VM_STAT_ADD(segvnvmstats.fullszcpages[9]);
3303         return (1);
3304 }

3306 /*
3307  * Create physically contiguous pages for [vp, off] - [vp, off +
3308  * page_size(szc)) range and for private segment return them in ppa array.
3309  * Pages are created either via IO or relocations.
3310  *
3311  * Return 1 on success and 0 on failure.
3312  *
3313  * If physically contiguous pages already exist for this range return 1 without
3314  * filling ppa array. Caller initializes ppa[0] as NULL to detect that ppa
3315  * array wasn't filled. In this case caller fills ppa array via VOP_GETPAGE().
3316  */

3318 static int
3319 segvn_fill_vp_pages(struct segvn_data *svd, vnode_t *vp, u_offset_t off,
3320     uint_t szc, page_t **ppa, page_t **ppplist, uint_t *ret_pszc,
3321     int *downsize)

3323 {
3324         page_t *pplist = *ppplist;
3325         size_t pgsz = page_get_pagesize(szc);
3326         pgcnt_t pages = btop(pgsz);
3327         ulong_t start_off = off;
3328         u_offset_t eoff = off + pgsz;
3329         spgcnt_t nreloc;
3330         u_offset_t io_off = off;
3331         size_t io_len;
3332         page_t *io_pplist = NULL;
3333         page_t *done_pplist = NULL;
3334         pgcnt_t pgidx = 0;
3335         page_t *pp;
3336         page_t *newpp;
3337         page_t *targpp;
3338         int io_err = 0;
3339         int i;
3340         pfn_t pfn;
3341         ulong_t ppages;
3342         page_t *targ_pplist = NULL;
3343         page_t *repl_pplist = NULL;
3344         page_t *tmp_pplist;
3345         int nios = 0;
3346         uint_t pszc;
3347         struct vattr va;

3349         VM_STAT_ADD(segvnvmstats.fill_vp_pages[0]);

3351         ASSERT(szc != 0);
3352         ASSERT(pplist->p_szc == szc);

3354         /*
3355          * downsize will be set to 1 only if we fail to lock pages. this will
3356          * allow subsequent faults to try to relocate the page again. If we
3357          * fail due to misalignment don't downsize and let the caller map the
3358          * whole region with small mappings to avoid more faults into the area
3359          * where we can't get large pages anyway.
3360          */
```



```
3361         *downsize = 0;

3363         while (off < eoff) {
3364                 newpp = pplist;
3365                 ASSERT(newpp != NULL);
3366                 ASSERT(PAGE_EXCL(newpp));
3367                 ASSERT(!PP_ISFREE(newpp));
3368                 /*
3369                  * we pass NULL for nrelocp to page_lookup_create()
3370                  * so that it doesn't relocate. We relocate here
3371                  * later only after we make sure we can lock all
3372                  * pages in the range we handle and they are all
3373                  * aligned.
3374                  */
3375                 pp = page_lookup_create(vp, off, SE_SHARED, newpp, NULL, 0);
3376                 ASSERT(pp != NULL);
3377                 ASSERT(!PP_ISFREE(pp));
3378                 ASSERT(pp->p_vnode == vp);
3379                 ASSERT(pp->p_offset == off);
3380                 if (pp == newpp) {
3381                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[1]);
3382                         page_sub(&pplist, pp);
3383                         ASSERT(PAGE_EXCL(pp));
3384                         ASSERT(page_iolock_assert(pp));
3385                         page_list_concat(&io_pplist, &pp);
3386                         off += PAGESIZE;
3387                         continue;
3388                 }
3389                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[2]);
3390                 pfn = page_pptonum(pp);
3391                 pszc = pp->p_szc;
3392                 if (pszc >= szc && targ_pplist == NULL && io_pplist == NULL &&
3393                     IS_P2ALIGNED(pfn, pages)) {
3394                         ASSERT(repl_pplist == NULL);
3395                         ASSERT(done_pplist == NULL);
3396                         ASSERT(pplist == *ppplist);
3397                         page_unlock(pp);
3398                         page_free_replacement_page(pplist);
3399                         page_create_putback(pages);
3400                         *ppplist = NULL;
3401                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[3]);
3402                         return (1);
3403                 }
3404                 if (pszc >= szc) {
3405                         page_unlock(pp);
3406                         segvn_faultvnmpss_align_err1++;
3407                         goto out;
3408                 }
3409                 ppages = page_get_pagecnt(pszc);
3410                 if (!IS_P2ALIGNED(pfn, ppages)) {
3411                         ASSERT(pszc > 0);
3412                         /*
3413                          * sizing down to pszc won't help.
3414                          */
3415                         page_unlock(pp);
3416                         segvn_faultvnmpss_align_err2++;
3417                         goto out;
3418                 }
3419                 pfn = page_pptonum(newpp);
3420                 if (!IS_P2ALIGNED(pfn, ppages)) {
3421                         ASSERT(pszc > 0);
3422                         /*
3423                          * sizing down to pszc won't help.
3424                          */
3425                         page_unlock(pp);
3426                         segvn_faultvnmpss_align_err3++;
```

```
3427                    goto out;
3428            }
3429            if (!PAGE_EXCL(pp)) {
3430                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[4]);
3431                    page_unlock(pp);
3432                    *downsize = 1;
3433                    *ret_pszc = pp->p_szc;
3434                    goto out;
3435            }
3436            targpp = pp;
3437            if (io_pplist != NULL) {
3438                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[5]);
3439                    io_len = off - io_off;
3440                    /*
3441                     * Some file systems like NFS don't check EOF
3442                     * conditions in VOP_PAGEIO(). Check it here
3443                     * now that pages are locked SE_EXCL. Any file
3444                     * truncation will wait until the pages are
3445                     * unlocked so no need to worry that file will
3446                     * be truncated after we check its size here.
3447                     * XXX fix NFS to remove this check.
3448                     */
3449                    va.va_mask = AT_SIZE;
3450                    if (VOP_GETATTR(vp, &va, ATTR_HINT, svd->cred, NULL)) {
3451                            VM_STAT_ADD(segvnvmstats.fill_vp_pages[6]);
3452                            page_unlock(targpp);
3453                            goto out;
3454                    }
3455                    if (btopr(va.va_size) < btopr(io_off + io_len)) {
3456                            VM_STAT_ADD(segvnvmstats.fill_vp_pages[7]);
3457                            *downsize = 1;
3458                            *ret_pszc = 0;
3459                            page_unlock(targpp);
3460                            goto out;
3461                    }
3462                    io_err = VOP_PAGEIO(vp, io_pplist, io_off, io_len,
3463                            B_READ, svd->cred, NULL);
3464                    if (io_err) {
3465                            VM_STAT_ADD(segvnvmstats.fill_vp_pages[8]);
3466                            page_unlock(targpp);
3467                            if (io_err == EDEADLK) {
3468                                    segvn_vmpss_pageio_deadlk_err++;
3469                            }
3470                            goto out;
3471                    }
3472                    nios++;
3473                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[9]);
3474                    while (io_pplist != NULL) {
3475                            pp = io_pplist;
3476                            page_sub(&io_pplist, pp);
3477                            ASSERT(page_iolock_assert(pp));
3478                            page_io_unlock(pp);
3479                            pgidx = (pp->p_offset - start_off) >>
3480                                PAGESHIFT;
3481                            ASSERT(pgidx < pages);
3482                            ppa[pgidx] = pp;
3483                            page_list_concat(&done_pplist, &pp);
3484                    }
3485            }
3486            pp = targpp;
3487            ASSERT(PAGE_EXCL(pp));
3488            ASSERT(pp->p_szc <= pszc);
3489            if (pszc != 0 && !group_page_trylock(pp, SE_EXCL)) {
3490                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[10]);
3491                    page_unlock(pp);
3492                    *downsize = 1;
```



```
3493                    *ret_pszc = pp->p_szc;
3494                    goto out;
3495            }
3496            VM_STAT_ADD(segvnvmstats.fill_vp_pages[11]);
3497            /*
3498             * page szc chould have changed before the entire group was
3499             * locked. reread page szc.
3500             */
3501            pszc = pp->p_szc;
3502            ppages = page_get_pagecnt(pszc);

3504            /* link just the roots */
3505            page_list_concat(&targ_pplist, &pp);
3506            page_sub(&pplist, newpp);
3507            page_list_concat(&repl_pplist, &newpp);
3508            off += PAGESIZE;
3509            while (--ppages != 0) {
3510                    newpp = pplist;
3511                    page_sub(&pplist, newpp);
3512                    off += PAGESIZE;
3513            }
3514            io_off = off;
3515    }
3516    if (io_pplist != NULL) {
3517            VM_STAT_ADD(segvnvmstats.fill_vp_pages[12]);
3518            io_len = eoff - io_off;
3519            va.va_mask = AT_SIZE;
3520            if (VOP_GETATTR(vp, &va, ATTR_HINT, svd->cred, NULL) != 0) {
3521                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[13]);
3522                    goto out;
3523            }
3524            if (btopr(va.va_size) < btopr(io_off + io_len)) {
3525                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[14]);
3526                    *downsize = 1;
3527                    *ret_pszc = 0;
3528                    goto out;
3529            }
3530            io_err = VOP_PAGEIO(vp, io_pplist, io_off, io_len,
3531                B_READ, svd->cred, NULL);
3532            if (io_err) {
3533                    VM_STAT_ADD(segvnvmstats.fill_vp_pages[15]);
3534                    if (io_err == EDEADLK) {
3535                            segvn_vmpss_pageio_deadlk_err++;
3536                    }
3537                    goto out;
3538            }
3539            nios++;
3540            while (io_pplist != NULL) {
3541                    pp = io_pplist;
3542                    page_sub(&io_pplist, pp);
3543                    ASSERT(page_iolock_assert(pp));
3544                    page_io_unlock(pp);
3545                    pgidx = (pp->p_offset - start_off) >> PAGESHIFT;
3546                    ASSERT(pgidx < pages);
3547                    ppa[pgidx] = pp;
3548            }
3549    }
3550    /*
3551     * we're now bound to succeed or panic.
3552     * remove pages from done_pplist. it's not needed anymore.
3553     */
3554    while (done_pplist != NULL) {
3555            pp = done_pplist;
3556            page_sub(&done_pplist, pp);
3557    }
3558    VM_STAT_ADD(segvnvmstats.fill_vp_pages[16]);
```

```
3559         ASSERT(pplist == NULL);
3560         *ppplist = NULL;
3561         while (targ_pplist != NULL) {
3562                 int ret;
3563                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[17]);
3564                 ASSERT(repl_pplist);
3565                 pp = targ_pplist;
3566                 page_sub(&targ_pplist, pp);
3567                 pgidx = (pp->p_offset - start_off) >> PAGESHIFT;
3568                 newpp = repl_pplist;
3569                 page_sub(&repl_pplist, newpp);
3570 #ifdef DEBUG
3571                 pfn = page_pptonum(pp);
3572                 pszc = pp->p_szc;
3573                 ppages = page_get_pagecnt(pszc);
3574                 ASSERT(IS_P2ALIGNED(pfn, ppages));
3575                 pfn = page_pptonum(newpp);
3576                 ASSERT(IS_P2ALIGNED(pfn, ppages));
3577                 ASSERT(P2PHASE(pfn, pages) == pgidx);
3578 #endif
3579                 nreloc = 0;
3580                 ret = page_relocate(&pp, &newpp, 0, 1, &nreloc, NULL);
3581                 if (ret != 0 || nreloc == 0) {
3582                         panic("segvn_fill_vp_pages: "
3583                             "page_relocate failed");
3584                 }
3585                 pp = newpp;
3586                 while (nreloc-- != 0) {
3587                         ASSERT(PAGE_EXCL(pp));
3588                         ASSERT(pp->p_vnode == vp);
3589                         ASSERT(pgidx ==
3590                             ((pp->p_offset - start_off) >> PAGESHIFT));
3591                         ppa[pgidx++] = pp;
3592                         pp++;
3593                 }
3594         }

3596         if (svd->type == MAP_PRIVATE) {
3597                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[18]);
3598                 for (i = 0; i < pages; i++) {
3599                         ASSERT(ppa[i] != NULL);
3600                         ASSERT(PAGE_EXCL(ppa[i]));
3601                         ASSERT(ppa[i]->p_vnode == vp);
3602                         ASSERT(ppa[i]->p_offset ==
3603                             start_off + (i << PAGESHIFT));
3604                         page_downgrade(ppa[i]);
3605                 }
3606                 ppa[pages] = NULL;
3607         } else {
3608                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[19]);
3609                 /*
3610                  * the caller will still call VOP_GETPAGE() for shared segments
3611                  * to check FS write permissions. For private segments we map
3612                  * file read only anyway.  so no VOP_GETPAGE is needed.
3613                  */
3614                 for (i = 0; i < pages; i++) {
3615                         ASSERT(ppa[i] != NULL);
3616                         ASSERT(PAGE_EXCL(ppa[i]));
3617                         ASSERT(ppa[i]->p_vnode == vp);
3618                         ASSERT(ppa[i]->p_offset ==
3619                             start_off + (i << PAGESHIFT));
3620                         page_unlock(ppa[i]);
3621                 }
3622                 ppa[0] = NULL;
3623         }
```



```
3625         return (1);
3626 out:
3627         /*
3628          * Do the cleanup. Unlock target pages we didn't relocate. They are
3629          * linked on targ_pplist by root pages. reassemble unused replacement
3630          * and io pages back to pplist.
3631          */
3632         if (io_pplist != NULL) {
3633                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[20]);
3634                 pp = io_pplist;
3635                 do {
3636                         ASSERT(pp->p_vnode == vp);
3637                         ASSERT(pp->p_offset == io_off);
3638                         ASSERT(page_iolock_assert(pp));
3639                         page_io_unlock(pp);
3640                         page_hashout(pp, NULL);
3641                         io_off += PAGESIZE;
3642                 } while ((pp = pp->p_next) != io_pplist);
3643                 page_list_concat(&io_pplist, &pplist);
3644                 pplist = io_pplist;
3645         }
3646         tmp_pplist = NULL;
3647         while (targ_pplist != NULL) {
3648                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[21]);
3649                 pp = targ_pplist;
3650                 ASSERT(PAGE_EXCL(pp));
3651                 page_sub(&targ_pplist, pp);

3653                 pszc = pp->p_szc;
3654                 ppages = page_get_pagecnt(pszc);
3655                 ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));

3657                 if (pszc != 0) {
3658                         group_page_unlock(pp);
3659                 }
3660                 page_unlock(pp);

3662                 pp = repl_pplist;
3663                 ASSERT(pp != NULL);
3664                 ASSERT(PAGE_EXCL(pp));
3665                 ASSERT(pp->p_szc == szc);
3666                 page_sub(&repl_pplist, pp);

3668                 ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));

3670                 /* relink replacement page */
3671                 page_list_concat(&tmp_pplist, &pp);
3672                 while (--ppages != 0) {
3673                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[22]);
3674                         pp++;
3675                         ASSERT(PAGE_EXCL(pp));
3676                         ASSERT(pp->p_szc == szc);
3677                         page_list_concat(&tmp_pplist, &pp);
3678                 }
3679         }
3680         if (tmp_pplist != NULL) {
3681                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[23]);
3682                 page_list_concat(&tmp_pplist, &pplist);
3683                 pplist = tmp_pplist;
3684         }
3685         /*
3686          * at this point all pages are either on done_pplist or
3687          * pplist. They can't be all on done_pplist otherwise
3688          * we'd've been done.
3689          */
3690         ASSERT(pplist != NULL);
```

```
3691         if (nios != 0) {
3692                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[24]);
3693                 pp = pplist;
3694                 do {
3695                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[25]);
3696                         ASSERT(pp->p_szc == szc);
3697                         ASSERT(PAGE_EXCL(pp));
3698                         ASSERT(pp->p_vnode != vp);
3699                         pp->p_szc = 0;
3700                 } while ((pp = pp->p_next) != pplist);

3702                 pp = done_pplist;
3703                 do {
3704                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[26]);
3705                         ASSERT(pp->p_szc == szc);
3706                         ASSERT(PAGE_EXCL(pp));
3707                         ASSERT(pp->p_vnode == vp);
3708                         pp->p_szc = 0;
3709                 } while ((pp = pp->p_next) != done_pplist);

3711                 while (pplist != NULL) {
3712                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[27]);
3713                         pp = pplist;
3714                         page_sub(&pplist, pp);
3715                         page_free(pp, 0);
3716                 }

3718                 while (done_pplist != NULL) {
3719                         VM_STAT_ADD(segvnvmstats.fill_vp_pages[28]);
3720                         pp = done_pplist;
3721                         page_sub(&done_pplist, pp);
3722                         page_unlock(pp);
3723                 }
3724                 *ppplist = NULL;
3725                 return (0);
3726         }
3727         ASSERT(pplist == *ppplist);
3728         if (io_err) {
3729                 VM_STAT_ADD(segvnvmstats.fill_vp_pages[29]);
3730                 /*
3731                  * don't downsize on io error.
3732                  * see if vop_getpage succeeds.
3733                  * pplist may still be used in this case
3734                  * for relocations.
3735                  */
3736                 return (0);
3737         }
3738         VM_STAT_ADD(segvnvmstats.fill_vp_pages[30]);
3739         page_free_replacement_page(pplist);
3740         page_create_putback(pages);
3741         *ppplist = NULL;
3742         return (0);
3743 }

3745 int segvn_anypgsz = 0;

3747 #define SEGVN_RESTORE_SOFTLOCK_VP(type, pages)                          \
3748                 if ((type) == F_SOFTLOCK) {                             \
3749                         atomic_add_long((ulong_t *)&(svd)->softlockcnt, \
3750                             -(pages));                                  \
3751                 }

3753 #define SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot)              \
3754                 if (IS_VMODSORT((ppa)[0]->p_vnode)) {                   \
3755                         if ((rw) == S_WRITE) {                          \
3756                                 for (i = 0; i < (pages); i++) {         \
```



```
3757                                         ASSERT((ppa)[i]->p_vnode ==     \
3758                                             (ppa)[0]->p_vnode);         \
3759                                         hat_setmod((ppa)[i]);           \
3760                                 }                                       \
3761                         } else if ((rw) != S_OTHER &&                   \
3762                             ((prot) & (vpprot) & PROT_WRITE)) {         \
3763                                 for (i = 0; i < (pages); i++) {         \
3764                                         ASSERT((ppa)[i]->p_vnode ==     \
3765                                             (ppa)[0]->p_vnode);         \
3766                                         if (!hat_ismod((ppa)[i])) {     \
3767                                                 prot &= ~PROT_WRITE;    \
3768                                                 break;                  \
3769                                         }                               \
3770                                 }                                       \
3771                         }                                               \
3772                 }

3774 #ifdef  VM_STATS

3776 #define SEGVN_VMSTAT_FLTVNPAGES(idx)                                    \
3777                 VM_STAT_ADD(segvnvmstats.fltvnpages[(idx)]);

3779 #else /* VM_STATS */

3781 #define SEGVN_VMSTAT_FLTVNPAGES(idx)

3783 #endif

3785 static faultcode_t
3786 segvn_fault_vnodepages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
3787     caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
3788     caddr_t eaddr, int brkcow)
3789 {
3790         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
3791         struct anon_map *amp = svd->amp;
3792         uchar_t segtype = svd->type;
3793         uint_t szc = seg->s_szc;
3794         size_t pgsz = page_get_pagesize(szc);
3795         size_t maxpgsz = pgsz;
3796         pgcnt_t pages = btop(pgsz);
3797         pgcnt_t maxpages = pages;
3798         size_t ppasize = (pages + 1) * sizeof (page_t *);
3799         caddr_t a = lpgaddr;
3800         caddr_t maxlpgeaddr = lpgeaddr;
3801         u_offset_t off = svd->offset + (uintptr_t)(a - seg->s_base);
3802         ulong_t aindx = svd->anon_index + seg_page(seg, a);
3803         struct vpage *vpage = (svd->vpage != NULL) ?
3804             &svd->vpage[seg_page(seg, a)] : NULL;
3805         vnode_t *vp = svd->vp;
3806         page_t **ppa;
3807         uint_t  pszc;
3808         size_t  ppgsz;
3809         pgcnt_t ppages;
3810         faultcode_t err = 0;
3811         int ierr;
3812         int vop_size_err = 0;
3813         uint_t protchk, prot, vpprot;
3814         ulong_t i;
3815         int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
3816         anon_sync_obj_t an_cookie;
3817         enum seg_rw arw;
3818         int alloc_failed = 0;
3819         int adjszc_chk;
3820         struct vattr va;
3821         int xhat = 0;
3822         page_t *pplist;
```

```
3823         pfn_t pfn;
3824         int physcontig;
3825         int upgrdfail;
3826         int segvn_anypgsz_vnode = 0; /* for now map vnode with 2 page sizes */
3827         int tron = (svd->tr_state == SEGVN_TR_ON);

3829         ASSERT(szc != 0);
3830         ASSERT(vp != NULL);
3831         ASSERT(brkcow == 0 || amp != NULL);
3832         ASSERT(tron == 0 || amp != NULL);
3833         ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
3834         ASSERT(!(svd->flags & MAP_NORESERVE));
3835         ASSERT(type != F_SOFTUNLOCK);
3836         ASSERT(IS_P2ALIGNED(a, maxpgsz));
3837         ASSERT(amp == NULL || IS_P2ALIGNED(aindx, maxpages));
3838         ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
3839         ASSERT(seg->s_szc < NBBY * sizeof (int));
3840         ASSERT(type != F_SOFTLOCK || lpgeaddr - a == maxpgsz);
3841         ASSERT(svd->tr_state != SEGVN_TR_INIT);

3843         VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnvmstats.fltvnpages[0]);
3844         VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnvmstats.fltvnpages[1]);

3846         if (svd->flags & MAP_TEXT) {
3847                 hat_flag |= HAT_LOAD_TEXT;
3848         }

3850         if (svd->pageprot) {
3851                 switch (rw) {
3852                 case S_READ:
3853                         protchk = PROT_READ;
3854                         break;
3855                 case S_WRITE:
3856                         protchk = PROT_WRITE;
3857                         break;
3858                 case S_EXEC:
3859                         protchk = PROT_EXEC;
3860                         break;
3861                 case S_OTHER:
3862                 default:
3863                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
3864                         break;
3865                 }
3866         } else {
3867                 prot = svd->prot;
3868                 /* caller has already done segment level protection check. */
3869         }

3871         if (seg->s_as->a_hat != hat) {
3872                 xhat = 1;
3873         }

3875         if (rw == S_WRITE && segtype == MAP_PRIVATE) {
3876                 SEGVN_VMSTAT_FLTVNPAGES(2);
3877                 arw = S_READ;
3878         } else {
3879                 arw = rw;
3880         }

3882         ppa = kmem_alloc(ppasize, KM_SLEEP);

3884         VM_STAT_COND_ADD(amp != NULL, segvnvmstats.fltvnpages[3]);

3886         for (;;) {
3887                 adjszc_chk = 0;
3888                 for (; a < lpgeaddr; a += pgsz, off += pgsz, aindx += pages) {
```



```
3889                         if (adjszc_chk) {
3890                                 while (szc < seg->s_szc) {
3891                                         uintptr_t e;
3892                                         uint_t tszc;
3893                                         tszc = segvn_anypgsz_vnode ? szc + 1 :
3894                                             seg->s_szc;
3895                                         ppgsz = page_get_pagesize(tszc);
3896                                         if (!IS_P2ALIGNED(a, ppgsz) ||
3897                                             ((alloc_failed >> tszc) & 0x1)) {
3898                                                 break;
3899                                         }
3900                                         SEGVN_VMSTAT_FLTVNPAGES(4);
3901                                         szc = tszc;
3902                                         pgsz = ppgsz;
3903                                         pages = btop(pgsz);
3904                                         e = P2ROUNDUP((uintptr_t)eaddr, pgsz);
3905                                         lpgeaddr = (caddr_t)e;
3906                                 }
3907                         }

3909                 again:
3910                         if (IS_P2ALIGNED(a, maxpgsz) && amp != NULL) {
3911                                 ASSERT(IS_P2ALIGNED(aindx, maxpages));
3912                                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
3913                                 anon_array_enter(amp, aindx, &an_cookie);
3914                                 if (anon_get_ptr(amp->ahp, aindx) != NULL) {
3915                                         SEGVN_VMSTAT_FLTVNPAGES(5);
3916                                         ASSERT(anon_pages(amp->ahp, aindx,
3917                                             maxpages) == maxpages);
3918                                         anon_array_exit(&an_cookie);
3919                                         ANON_LOCK_EXIT(&amp->a_rwlock);
3920                                         err = segvn_fault_anonpages(hat, seg,
3921                                             a, a + maxpgsz, type, rw,
3922                                             MAX(a, addr),
3923                                             MIN(a + maxpgsz, eaddr), brkcow);
3924                                         if (err != 0) {
3925                                                 SEGVN_VMSTAT_FLTVNPAGES(6);
3926                                                 goto out;
3927                                         }
3928                                         if (szc < seg->s_szc) {
3929                                                 szc = seg->s_szc;
3930                                                 pgsz = maxpgsz;
3931                                                 pages = maxpages;
3932                                                 lpgeaddr = maxlpgeaddr;
3933                                         }
3934                                         goto next;
3935                                 } else {
3936                                         ASSERT(anon_pages(amp->ahp, aindx,
3937                                             maxpages) == 0);
3938                                         SEGVN_VMSTAT_FLTVNPAGES(7);
3939                                         anon_array_exit(&an_cookie);
3940                                         ANON_LOCK_EXIT(&amp->a_rwlock);
3941                                 }
3942                         }
3943                         ASSERT(!brkcow || IS_P2ALIGNED(a, maxpgsz));
3944                         ASSERT(!tron || IS_P2ALIGNED(a, maxpgsz));

3946                         if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
3947                                 ASSERT(vpage != NULL);
3948                                 prot = VPP_PROT(vpage);
3949                                 ASSERT(sameprot(seg, a, maxpgsz));
3950                                 if ((prot & protchk) == 0) {
3951                                         SEGVN_VMSTAT_FLTVNPAGES(8);
3952                                         err = FC_PROT;
3953                                         goto out;
3954                                 }
```

```
3955                        }
3956                        if (type == F_SOFTLOCK) {
3957                                atomic_add_long((ulong_t *)&svd->softlockcnt,
3958                                    pages);
3959                        }

3961                        pplist = NULL;
3962                        physcontig = 0;
3963                        ppa[0] = NULL;
3964                        if (!brkcow && !tron && szc &&
3965                            !page_exists_physcontig(vp, off, szc,
3966                            segtype == MAP_PRIVATE ? ppa : NULL)) {
3967                                SEGVN_VMSTAT_FLTVNPAGES(9);
3968                                if (page_alloc_pages(vp, seg, a, &pplist, NULL,
3969                                    szc, 0, 0) && type != F_SOFTLOCK) {
3970                                        SEGVN_VMSTAT_FLTVNPAGES(10);
3971                                        pszc = 0;
3972                                        ierr = -1;
3973                                        alloc_failed |= (1 << szc);
3974                                        break;
3975                                }
3976                                if (pplist != NULL &&
3977                                    vp->v_mpssdata == SEGVN_PAGEIO) {
3978                                        int downsize;
3979                                        SEGVN_VMSTAT_FLTVNPAGES(11);
3980                                        physcontig = segvn_fill_vp_pages(svd,
3981                                            vp, off, szc, ppa, &pplist,
3982                                            &pszc, &downsize);
3983                                        ASSERT(!physcontig || pplist == NULL);
3984                                        if (!physcontig && downsize &&
3985                                            type != F_SOFTLOCK) {
3986                                                ASSERT(pplist == NULL);
3987                                                SEGVN_VMSTAT_FLTVNPAGES(12);
3988                                                ierr = -1;
3989                                                break;
3990                                        }
3991                                        ASSERT(!physcontig ||
3992                                            segtype == MAP_PRIVATE ||
3993                                            ppa[0] == NULL);
3994                                        if (physcontig && ppa[0] == NULL) {
3995                                                physcontig = 0;
3996                                        }
3997                                }
3998                        } else if (!brkcow && !tron && szc && ppa[0] != NULL) {
3999                                SEGVN_VMSTAT_FLTVNPAGES(13);
4000                                ASSERT(segtype == MAP_PRIVATE);
4001                                physcontig = 1;
4002                        }

4004                        if (!physcontig) {
4005                                SEGVN_VMSTAT_FLTVNPAGES(14);
4006                                ppa[0] = NULL;
4007                                ierr = VOP_GETPAGE(vp, (offset_t)off, pgsz,
4008                                    &vpprot, ppa, pgsz, seg, a, arw,
4009                                    svd->cred, NULL);
4010 #ifdef DEBUG
4011                                if (ierr == 0) {
4012                                        for (i = 0; i < pages; i++) {
4013                                                ASSERT(PAGE_LOCKED(ppa[i]));
4014                                                ASSERT(!PP_ISFREE(ppa[i]));
4015                                                ASSERT(ppa[i]->p_vnode == vp);
4016                                                ASSERT(ppa[i]->p_offset ==
4017                                                    off + (i << PAGESHIFT));
4018                                        }
4019                                }
4020 #endif /* DEBUG */
```



```
4021                                if (segtype == MAP_PRIVATE) {
4022                                        SEGVN_VMSTAT_FLTVNPAGES(15);
4023                                        vpprot &= ~PROT_WRITE;
4024                                }
4025                        } else {
4026                                ASSERT(segtype == MAP_PRIVATE);
4027                                SEGVN_VMSTAT_FLTVNPAGES(16);
4028                                vpprot = PROT_ALL & ~PROT_WRITE;
4029                                ierr = 0;
4030                        }

4032                        if (ierr != 0) {
4033                                SEGVN_VMSTAT_FLTVNPAGES(17);
4034                                if (pplist != NULL) {
4035                                        SEGVN_VMSTAT_FLTVNPAGES(18);
4036                                        page_free_replacement_page(pplist);
4037                                        page_create_putback(pages);
4038                                }
4039                                SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4040                                if (a + pgsz <= eaddr) {
4041                                        SEGVN_VMSTAT_FLTVNPAGES(19);
4042                                        err = FC_MAKE_ERR(ierr);
4043                                        goto out;
4044                                }
4045                                va.va_mask = AT_SIZE;
4046                                if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL)) {
4047                                        SEGVN_VMSTAT_FLTVNPAGES(20);
4048                                        err = FC_MAKE_ERR(EIO);
4049                                        goto out;
4050                                }
4051                                if (btopr(va.va_size) >= btopr(off + pgsz)) {
4052                                        SEGVN_VMSTAT_FLTVNPAGES(21);
4053                                        err = FC_MAKE_ERR(ierr);
4054                                        goto out;
4055                                }
4056                                if (btopr(va.va_size) <
4057                                    btopr(off + (eaddr - a))) {
4058                                        SEGVN_VMSTAT_FLTVNPAGES(22);
4059                                        err = FC_MAKE_ERR(ierr);
4060                                        goto out;
4061                                }
4062                                if (brkcow || tron || type == F_SOFTLOCK) {
4063                                        /* can't reduce map area */
4064                                        SEGVN_VMSTAT_FLTVNPAGES(23);
4065                                        vop_size_err = 1;
4066                                        goto out;
4067                                }
4068                                SEGVN_VMSTAT_FLTVNPAGES(24);
4069                                ASSERT(szc != 0);
4070                                pszc = 0;
4071                                ierr = -1;
4072                                break;
4073                        }

4075                        if (amp != NULL) {
4076                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
4077                                anon_array_enter(amp, aindx, &an_cookie);
4078                        }
4079                        if (amp != NULL &&
4080                            anon_get_ptr(amp->ahp, aindx) != NULL) {
4081                                ulong_t taindx = P2ALIGN(aindx, maxpages);

4083                                SEGVN_VMSTAT_FLTVNPAGES(25);
4084                                ASSERT(anon_pages(amp->ahp, taindx,
4085                                    maxpages) == maxpages);
4086                                for (i = 0; i < pages; i++) {
```

```
4087                                    page_unlock(ppa[i]);
4088                            }
4089                            anon_array_exit(&an_cookie);
4090                            ANON_LOCK_EXIT(&amp->a_rwlock);
4091                            if (pplist != NULL) {
4092                                    page_free_replacement_page(pplist);
4093                                    page_create_putback(pages);
4094                            }
4095                            SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4096                            if (szc < seg->s_szc) {
4097                                    SEGVN_VMSTAT_FLTVNPAGES(26);
4098                                    /*
4099                                     * For private segments SOFTLOCK
4100                                     * either always breaks cow (any rw
4101                                     * type except S_READ_NOCOW) or
4102                                     * address space is locked as writer
4103                                     * (S_READ_NOCOW case) and anon slots
4104                                     * can't show up on second check.
4105                                     * Therefore if we are here for
4106                                     * SOFTLOCK case it must be a cow
4107                                     * break but cow break never reduces
4108                                     * szc. text replication (tron) in
4109                                     * this case works as cow break.
4110                                     * Thus the assert below.
4111                                     */
4112                                    ASSERT(!brkcow && !tron &&
4113                                        type != F_SOFTLOCK);
4114                                    pszc = seg->s_szc;
4115                                    ierr = -2;
4116                                    break;
4117                            }
4118                            ASSERT(IS_P2ALIGNED(a, maxpgsz));
4119                            goto again;
4120                    }
4121 #ifdef DEBUG
4122                    if (amp != NULL) {
4123                            ulong_t taindx = P2ALIGN(aindx, maxpages);
4124                            ASSERT(!anon_pages(amp->ahp, taindx, maxpages));
4125                    }
4126 #endif /* DEBUG */
4127
4128                    if (brkcow || tron) {
4129                            ASSERT(amp != NULL);
4130                            ASSERT(pplist == NULL);
4131                            ASSERT(szc == seg->s_szc);
4132                            ASSERT(IS_P2ALIGNED(a, maxpgsz));
4133                            ASSERT(IS_P2ALIGNED(aindx, maxpages));
4134                            SEGVN_VMSTAT_FLTVNPAGES(27);
4135                            ierr = anon_map_privatepages(amp, aindx, szc,
4136                                seg, a, prot, ppa, vpage, segvn_anypgsz,
4137                                tron ? PG_LOCAL : 0, svd->cred);
4138                            if (ierr != 0) {
4139                                    SEGVN_VMSTAT_FLTVNPAGES(28);
4140                                    anon_array_exit(&an_cookie);
4141                                    ANON_LOCK_EXIT(&amp->a_rwlock);
4142                                    SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4143                                    err = FC_MAKE_ERR(ierr);
4144                                    goto out;
4145                            }
4146
4147                            ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));
4148                            /*
4149                             * p_szc can't be changed for locked
4150                             * swapfs pages.
4151                             */
4152                            ASSERT(svd->rcookie ==
```



```
4153                                HAT_INVALID_REGION_COOKIE);
4154                            hat_memload_array(hat, a, pgsz, ppa, prot,
4155                                hat_flag);
4156
4157                            if (!(hat_flag & HAT_LOAD_LOCK)) {
4158                                    SEGVN_VMSTAT_FLTVNPAGES(29);
4159                                    for (i = 0; i < pages; i++) {
4160                                            page_unlock(ppa[i]);
4161                                    }
4162                            }
4163                            anon_array_exit(&an_cookie);
4164                            ANON_LOCK_EXIT(&amp->a_rwlock);
4165                            goto next;
4166                    }
4167
4168                    ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
4169                        (!svd->pageprot && svd->prot == (prot & vpprot)));
4170
4171                    pfn = page_pptonum(ppa[0]);
4172                    /*
4173                     * hat_page_demote() needs an SE_EXCL lock on one of
4174                     * constituent page_t's and it decreases root's p_szc
4175                     * last. This means if root's p_szc is equal szc and
4176                     * all its constituent pages are locked
4177                     * hat_page_demote() that could have changed p_szc to
4178                     * szc is already done and no new have page_demote()
4179                     * can start for this large page.
4180                     */
4181
4182                    /*
4183                     * we need to make sure same mapping size is used for
4184                     * the same address range if there's a possibility the
4185                     * adddress is already mapped because hat layer panics
4186                     * when translation is loaded for the range already
4187                     * mapped with a different page size.  We achieve it
4188                     * by always using largest page size possible subject
4189                     * to the constraints of page size, segment page size
4190                     * and page alignment.  Since mappings are invalidated
4191                     * when those constraints change and make it
4192                     * impossible to use previously used mapping size no
4193                     * mapping size conflicts should happen.
4194                     */
4195
4196            chkszc:
4197                    if ((pszc = ppa[0]->p_szc) == szc &&
4198                        IS_P2ALIGNED(pfn, pages)) {
4199
4200                            SEGVN_VMSTAT_FLTVNPAGES(30);
4201 #ifdef DEBUG
4202                            for (i = 0; i < pages; i++) {
4203                                    ASSERT(PAGE_LOCKED(ppa[i]));
4204                                    ASSERT(!PP_ISFREE(ppa[i]));
4205                                    ASSERT(page_pptonum(ppa[i]) ==
4206                                        pfn + i);
4207                                    ASSERT(ppa[i]->p_szc == szc);
4208                                    ASSERT(ppa[i]->p_vnode == vp);
4209                                    ASSERT(ppa[i]->p_offset ==
4210                                        off + (i << PAGESHIFT));
4211                            }
4212 #endif /* DEBUG */
4213                            /*
4214                             * All pages are of szc we need and they are
4215                             * all locked so they can't change szc. load
4216                             * translations.
4217                             *
4218                             * if page got promoted since last check
```

```
4219                                 * we don't need pplist.
4220                                 */
4221                                if (pplist != NULL) {
4222                                        page_free_replacement_page(pplist);
4223                                        page_create_putback(pages);
4224                                }
4225                                if (PP_ISMIGRATE(ppa[0])) {
4226                                        page_migrate(seg, a, ppa, pages);
4227                                }
4228                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4229                                    prot, vpprot);
4230                                if (!xhat) {
4231                                        hat_memload_array_region(hat, a, pgsz,
4232                                            ppa, prot & vpprot, hat_flag,
4233                                            svd->rcookie);
4234                                } else {
4235                                        /*
4236                                         * avoid large xhat mappings to FS
4237                                         * pages so that hat_page_demote()
4238                                         * doesn't need to check for xhat
4239                                         * large mappings.
4240                                         * Don't use regions with xhats.
4241                                         */
4242                                        for (i = 0; i < pages; i++) {
4243                                                hat_memload(hat,
4244                                                    a + (i << PAGESHIFT),
4245                                                    ppa[i], prot & vpprot,
4246                                                    hat_flag);
4247                                        }
4248                                }

4250                                if (!(hat_flag & HAT_LOAD_LOCK)) {
4251                                        for (i = 0; i < pages; i++) {
4252                                                page_unlock(ppa[i]);
4253                                        }
4254                                }
4255                                if (amp != NULL) {
4256                                        anon_array_exit(&an_cookie);
4257                                        ANON_LOCK_EXIT(&amp->a_rwlock);
4258                                }
4259                                goto next;
4260                        }

4262                        /*
4263                         * See if upsize is possible.
4264                         */
4265                        if (pszc > szc && szc < seg->s_szc &&
4266                            (segvn_anypgsz_vnode || pszc >= seg->s_szc)) {
4267                                pgcnt_t aphase;
4268                                uint_t pszc1 = MIN(pszc, seg->s_szc);
4269                                ppgsz = page_get_pagesize(pszc1);
4270                                ppages = btop(ppgsz);
4271                                aphase = btop(P2PHASE((uintptr_t)a, ppgsz));

4273                                ASSERT(type != F_SOFTLOCK);

4275                                SEGVN_VMSTAT_FLTVNPAGES(31);
4276                                if (aphase != P2PHASE(pfn, ppages)) {
4277                                        segvn_faultvnmpss_align_err4++;
4278                                } else {
4279                                        SEGVN_VMSTAT_FLTVNPAGES(32);
4280                                        if (pplist != NULL) {
4281                                                page_t *pl = pplist;
4282                                                page_free_replacement_page(pl);
4283                                                page_create_putback(pages);
4284                                        }
```



```
4285                                        for (i = 0; i < pages; i++) {
4286                                                page_unlock(ppa[i]);
4287                                        }
4288                                        if (amp != NULL) {
4289                                                anon_array_exit(&an_cookie);
4290                                                ANON_LOCK_EXIT(&amp->a_rwlock);
4291                                        }
4292                                        pszc = pszc1;
4293                                        ierr = -2;
4294                                        break;
4295                                }
4296                        }

4298                        /*
4299                         * check if we should use smallest mapping size.
4300                         */
4301                        upgrdfail = 0;
4302                        if (szc == 0 || xhat ||
4303                            (pszc >= szc &&
4304                            !IS_P2ALIGNED(pfn, pages)) ||
4305                            (pszc < szc &&
4306                            !segvn_full_szcpages(ppa, szc, &upgrdfail,
4307                            &pszc))) {

4309                                if (upgrdfail && type != F_SOFTLOCK) {
4310                                        /*
4311                                         * segvn_full_szcpages failed to lock
4312                                         * all pages EXCL. Size down.
4313                                         */
4314                                        ASSERT(pszc < szc);

4316                                        SEGVN_VMSTAT_FLTVNPAGES(33);

4318                                        if (pplist != NULL) {
4319                                                page_t *pl = pplist;
4320                                                page_free_replacement_page(pl);
4321                                                page_create_putback(pages);
4322                                        }

4324                                        for (i = 0; i < pages; i++) {
4325                                                page_unlock(ppa[i]);
4326                                        }
4327                                        if (amp != NULL) {
4328                                                anon_array_exit(&an_cookie);
4329                                                ANON_LOCK_EXIT(&amp->a_rwlock);
4330                                        }
4331                                        ierr = -1;
4332                                        break;
4333                                }
4334                                if (szc != 0 && !xhat && !upgrdfail) {
4335                                        segvn_faultvnmpss_align_err5++;
4336                                }
4337                                SEGVN_VMSTAT_FLTVNPAGES(34);
4338                                if (pplist != NULL) {
4339                                        page_free_replacement_page(pplist);
4340                                        page_create_putback(pages);
4341                                }
4342                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4343                                    prot, vpprot);
4344                                if (upgrdfail && segvn_anypgsz_vnode) {
4345                                        /* SOFTLOCK case */
4346                                        hat_memload_array_region(hat, a, pgsz,
4347                                            ppa, prot & vpprot, hat_flag,
4348                                            svd->rcookie);
4349                                } else {
4350                                        for (i = 0; i < pages; i++) {
```

```
4351                                    hat_memload_region(hat,
4352                                        a + (i << PAGESHIFT),
4353                                        ppa[i], prot & vpprot,
4354                                        hat_flag, svd->rcookie);
4355                            }
4356                    }
4357                    if (!(hat_flag & HAT_LOAD_LOCK)) {
4358                            for (i = 0; i < pages; i++) {
4359                                    page_unlock(ppa[i]);
4360                            }
4361                    }
4362                    if (amp != NULL) {
4363                            anon_array_exit(&an_cookie);
4364                            ANON_LOCK_EXIT(&amp->a_rwlock);
4365                    }
4366                    goto next;
4367            }

4369            if (pszc == szc) {
4370                    /*
4371                     * segvn_full_szcpages() upgraded pages szc.
4372                     */
4373                    ASSERT(pszc == ppa[0]->p_szc);
4374                    ASSERT(IS_P2ALIGNED(pfn, pages));
4375                    goto chkszc;
4376            }

4378            if (pszc > szc) {
4379                    kmutex_t *szcmtx;
4380                    SEGVN_VMSTAT_FLTVNPAGES(35);
4381                    /*
4382                     * p_szc of ppa[0] can change since we haven't
4383                     * locked all constituent pages. Call
4384                     * page_lock_szc() to prevent szc changes.
4385                     * This should be a rare case that happens when
4386                     * multiple segments use a different page size
4387                     * to map the same file offsets.
4388                     */
4389                    szcmtx = page_szc_lock(ppa[0]);
4390                    pszc = ppa[0]->p_szc;
4391                    ASSERT(szcmtx != NULL || pszc == 0);
4392                    ASSERT(ppa[0]->p_szc <= pszc);
4393                    if (pszc <= szc) {
4394                            SEGVN_VMSTAT_FLTVNPAGES(36);
4395                            if (szcmtx != NULL) {
4396                                    mutex_exit(szcmtx);
4397                            }
4398                            goto chkszc;
4399                    }
4400                    if (pplist != NULL) {
4401                            /*
4402                             * page got promoted since last check.
4403                             * we don't need preaalocated large
4404                             * page.
4405                             */
4406                            SEGVN_VMSTAT_FLTVNPAGES(37);
4407                            page_free_replacement_page(pplist);
4408                            page_create_putback(pages);
4409                    }
4410                    SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4411                        prot, vpprot);
4412                    hat_memload_array_region(hat, a, pgsz, ppa,
4413                        prot & vpprot, hat_flag, svd->rcookie);
4414                    mutex_exit(szcmtx);
4415                    if (!(hat_flag & HAT_LOAD_LOCK)) {
4416                            for (i = 0; i < pages; i++) {
```



```
4417                                    page_unlock(ppa[i]);
4418                            }
4419                    }
4420                    if (amp != NULL) {
4421                            anon_array_exit(&an_cookie);
4422                            ANON_LOCK_EXIT(&amp->a_rwlock);
4423                    }
4424                    goto next;
4425            }

4427            /*
4428             * if page got demoted since last check
4429             * we could have not allocated larger page.
4430             * allocate now.
4431             */
4432            if (pplist == NULL &&
4433                page_alloc_pages(vp, seg, a, &pplist, NULL,
4434                szc, 0, 0) && type != F_SOFTLOCK) {
4435                    SEGVN_VMSTAT_FLTVNPAGES(38);
4436                    for (i = 0; i < pages; i++) {
4437                            page_unlock(ppa[i]);
4438                    }
4439                    if (amp != NULL) {
4440                            anon_array_exit(&an_cookie);
4441                            ANON_LOCK_EXIT(&amp->a_rwlock);
4442                    }
4443                    ierr = -1;
4444                    alloc_failed |= (1 << szc);
4445                    break;
4446            }

4448            SEGVN_VMSTAT_FLTVNPAGES(39);

4450            if (pplist != NULL) {
4451                    segvn_relocate_pages(ppa, pplist);
4452 #ifdef DEBUG
4453            } else {
4454                    ASSERT(type == F_SOFTLOCK);
4455                    SEGVN_VMSTAT_FLTVNPAGES(40);
4456 #endif /* DEBUG */
4457            }

4459            SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot);

4461            if (pplist == NULL && segvn_anypgsz_vnode == 0) {
4462                    ASSERT(type == F_SOFTLOCK);
4463                    for (i = 0; i < pages; i++) {
4464                            ASSERT(ppa[i]->p_szc < szc);
4465                            hat_memload_region(hat,
4466                                a + (i << PAGESHIFT),
4467                                ppa[i], prot & vpprot, hat_flag,
4468                                svd->rcookie);
4469                    }
4470            } else {
4471                    ASSERT(pplist != NULL || type == F_SOFTLOCK);
4472                    hat_memload_array_region(hat, a, pgsz, ppa,
4473                        prot & vpprot, hat_flag, svd->rcookie);
4474            }
4475            if (!(hat_flag & HAT_LOAD_LOCK)) {
4476                    for (i = 0; i < pages; i++) {
4477                            ASSERT(PAGE_SHARED(ppa[i]));
4478                            page_unlock(ppa[i]);
4479                    }
4480            }
4481            if (amp != NULL) {
4482                    anon_array_exit(&an_cookie);
```

```
4483                                    ANON_LOCK_EXIT(&amp->a_rwlock);
4484                            }
4485
4486                    next:
4487                            if (vpage != NULL) {
4488                                    vpage += pages;
4489                            }
4490                            adjszc_chk = 1;
4491                    }
4492                    if (a == lpgeaddr)
4493                            break;
4494                    ASSERT(a < lpgeaddr);
4495
4496                    ASSERT(!brkcow && !tron && type != F_SOFTLOCK);
4497
4498                    /*
4499                     * ierr == -1 means we failed to map with a large page.
4500                     * (either due to allocation/relocation failures or
4501                     * misalignment with other mappings to this file.
4502                     *
4503                     * ierr == -2 means some other thread allocated a large page
4504                     * after we gave up tp map with a large page.  retry with
4505                     * larger mapping.
4506                     */
4507                    ASSERT(ierr == -1 || ierr == -2);
4508                    ASSERT(ierr == -2 || szc != 0);
4509                    ASSERT(ierr == -1 || szc < seg->s_szc);
4510                    if (ierr == -2) {
4511                            SEGVN_VMSTAT_FLTVNPAGES(41);
4512                            ASSERT(pszc > szc && pszc <= seg->s_szc);
4513                            szc = pszc;
4514                    } else if (segvn_anypgsz_vnode) {
4515                            SEGVN_VMSTAT_FLTVNPAGES(42);
4516                            szc--;
4517                    } else {
4518                            SEGVN_VMSTAT_FLTVNPAGES(43);
4519                            ASSERT(pszc < szc);
4520                            /*
4521                             * other process created pszc large page.
4522                             * but we still have to drop to 0 szc.
4523                             */
4524                            szc = 0;
4525                    }
4526
4527                    pgsz = page_get_pagesize(szc);
4528                    pages = btop(pgsz);
4529                    if (ierr == -2) {
4530                            /*
4531                             * Size up case. Note lpgaddr may only be needed for
4532                             * softlock case so we don't adjust it here.
4533                             */
4534                            a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
4535                            ASSERT(a >= lpgaddr);
4536                            lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4537                            off = svd->offset + (uintptr_t)(a - seg->s_base);
4538                            aindx = svd->anon_index + seg_page(seg, a);
4539                            vpage = (svd->vpage != NULL) ?
4540                                &svd->vpage[seg_page(seg, a)] : NULL;
4541                    } else {
4542                            /*
4543                             * Size down case. Note lpgaddr may only be needed for
4544                             * softlock case so we don't adjust it here.
4545                             */
4546                            ASSERT(IS_P2ALIGNED(a, pgsz));
4547                            ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
4548                            lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
```



```
4549                            ASSERT(a < lpgeaddr);
4550                            if (a < addr) {
4551                                    SEGVN_VMSTAT_FLTVNPAGES(44);
4552                                    /*
4553                                     * The beginning of the large page region can
4554                                     * be pulled to the right to make a smaller
4555                                     * region. We haven't yet faulted a single
4556                                     * page.
4557                                     */
4558                                    a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
4559                                    ASSERT(a >= lpgaddr);
4560                                    off = svd->offset +
4561                                        (uintptr_t)(a - seg->s_base);
4562                                    aindx = svd->anon_index + seg_page(seg, a);
4563                                    vpage = (svd->vpage != NULL) ?
4564                                        &svd->vpage[seg_page(seg, a)] : NULL;
4565                            }
4566                    }
4567            }
4568    out:
4569            kmem_free(ppa, ppasize);
4570            if (!err && !vop_size_err) {
4571                    SEGVN_VMSTAT_FLTVNPAGES(45);
4572                    return (0);
4573            }
4574            if (type == F_SOFTLOCK && a > lpgaddr) {
4575                    SEGVN_VMSTAT_FLTVNPAGES(46);
4576                    segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4577            }
4578            if (!vop_size_err) {
4579                    SEGVN_VMSTAT_FLTVNPAGES(47);
4580                    return (err);
4581            }
4582            ASSERT(brkcow || tron || type == F_SOFTLOCK);
4583            /*
4584             * Large page end is mapped beyond the end of file and it's a cow
4585             * fault (can be a text replication induced cow) or softlock so we can't
4586             * reduce the map area.  For now just demote the segment. This should
4587             * really only happen if the end of the file changed after the mapping
4588             * was established since when large page segments are created we make
4589             * sure they don't extend beyond the end of the file.
4590             */
4591            SEGVN_VMSTAT_FLTVNPAGES(48);
4592
4593            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4594            SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4595            err = 0;
4596            if (seg->s_szc != 0) {
4597                    segvn_fltvnpages_clrszc_cnt++;
4598                    ASSERT(svd->softlockcnt == 0);
4599                    err = segvn_clrszc(seg);
4600                    if (err != 0) {
4601                            segvn_fltvnpages_clrszc_err++;
4602                    }
4603            }
4604            ASSERT(err || seg->s_szc == 0);
4605            SEGVN_LOCK_DOWNGRADE(seg->s_as, &svd->lock);
4606            /* segvn_fault will do its job as if szc had been zero to begin with */
4607            return (err == 0 ? IE_RETRY : FC_MAKE_ERR(err));
4608    }
4609
4610    /*
4611     * This routine will attempt to fault in one large page.
4612     * it will use smaller pages if that fails.
4613     * It should only be called for pure anonymous segments.
4614     */
```

```
4615 static faultcode_t
4616 segvn_fault_anonpages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
4617     caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
4618     caddr_t eaddr, int brkcow)
4619 {
4620         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4621         struct anon_map *amp = svd->amp;
4622         uchar_t segtype = svd->type;
4623         uint_t szc = seg->s_szc;
4624         size_t pgsz = page_get_pagesize(szc);
4625         size_t maxpgsz = pgsz;
4626         pgcnt_t pages = btop(pgsz);
4627         uint_t ppaszc = szc;
4628         caddr_t a = lpgaddr;
4629         ulong_t aindx = svd->anon_index + seg_page(seg, a);
4630         struct vpage *vpage = (svd->vpage != NULL) ?
4631             &svd->vpage[seg_page(seg, a)] : NULL;
4632         page_t **ppa;
4633         uint_t  ppa_szc;
4634         faultcode_t err;
4635         int ierr;
4636         uint_t protchk, prot, vpprot;
4637         ulong_t i;
4638         int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
4639         anon_sync_obj_t cookie;
4640         int adjszc_chk;
4641         int pgflags = (svd->tr_state == SEGVN_TR_ON) ? PG_LOCAL : 0;

4643         ASSERT(szc != 0);
4644         ASSERT(amp != NULL);
4645         ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
4646         ASSERT(!(svd->flags & MAP_NORESERVE));
4647         ASSERT(type != F_SOFTUNLOCK);
4648         ASSERT(IS_P2ALIGNED(a, maxpgsz));
4649         ASSERT(!brkcow || svd->tr_state == SEGVN_TR_OFF);
4650         ASSERT(svd->tr_state != SEGVN_TR_INIT);

4652         ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

4654         VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnvmstats.fltanpages[0]);
4655         VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnvmstats.fltanpages[1]);

4657         if (svd->flags & MAP_TEXT) {
4658                 hat_flag |= HAT_LOAD_TEXT;
4659         }

4661         if (svd->pageprot) {
4662                 switch (rw) {
4663                 case S_READ:
4664                         protchk = PROT_READ;
4665                         break;
4666                 case S_WRITE:
4667                         protchk = PROT_WRITE;
4668                         break;
4669                 case S_EXEC:
4670                         protchk = PROT_EXEC;
4671                         break;
4672                 case S_OTHER:
4673                 default:
4674                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
4675                         break;
4676                 }
4677                 VM_STAT_ADD(segvnvmstats.fltanpages[2]);
4678         } else {
4679                 prot = svd->prot;
4680                 /* caller has already done segment level protection check. */
```



```
4681         }

4683         ppa = kmem_cache_alloc(segvn_szc_cache[ppaszc], KM_SLEEP);
4684         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
4685         for (;;) {
4686                 adjszc_chk = 0;
4687                 for (; a < lpgeaddr; a += pgsz, aindx += pages) {
4688                         if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
4689                                 VM_STAT_ADD(segvnvmstats.fltanpages[3]);
4690                                 ASSERT(vpage != NULL);
4691                                 prot = VPP_PROT(vpage);
4692                                 ASSERT(sameprot(seg, a, maxpgsz));
4693                                 if ((prot & protchk) == 0) {
4694                                         err = FC_PROT;
4695                                         goto error;
4696                                 }
4697                         }
4698                         if (adjszc_chk && IS_P2ALIGNED(a, maxpgsz) &&
4699                             pgsz < maxpgsz) {
4700                                 ASSERT(a > lpgaddr);
4701                                 szc = seg->s_szc;
4702                                 pgsz = maxpgsz;
4703                                 pages = btop(pgsz);
4704                                 ASSERT(IS_P2ALIGNED(aindx, pages));
4705                                 lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr,
4706                                     pgsz);
4707                         }
4708                         if (type == F_SOFTLOCK) {
4709                                 atomic_add_long((ulong_t *)&svd->softlockcnt,
4710                                     pages);
4711                         }
4712                         anon_array_enter(amp, aindx, &cookie);
4713                         ppa_szc = (uint_t)-1;
4714                         ierr = anon_map_getpages(amp, aindx, szc, seg, a,
4715                             prot, &vpprot, ppa, &ppa_szc, vpage, rw, brkcow,
4716                             segvn_anypgsz, pgflags, svd->cred);
4717                         if (ierr != 0) {
4718                                 anon_array_exit(&cookie);
4719                                 VM_STAT_ADD(segvnvmstats.fltanpages[4]);
4720                                 if (type == F_SOFTLOCK) {
4721                                         atomic_add_long(
4722                                             (ulong_t *)&svd->softlockcnt,
4723                                             -pages);
4724                                 }
4725                                 if (ierr > 0) {
4726                                         VM_STAT_ADD(segvnvmstats.fltanpages[6]);
4727                                         err = FC_MAKE_ERR(ierr);
4728                                         goto error;
4729                                 }
4730                                 break;
4731                         }

4733                         ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));

4735                         ASSERT(segtype == MAP_SHARED ||
4736                             ppa[0]->p_szc <= szc);
4737                         ASSERT(segtype == MAP_PRIVATE ||
4738                             ppa[0]->p_szc >= szc);

4740                         /*
4741                          * Handle pages that have been marked for migration
4742                          */
4743                         if (lgrp_optimizations())
4744                                 page_migrate(seg, a, ppa, pages);

4746                         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
```

```
4748                        if (segtype == MAP_SHARED) {
4749                                vpprot |= PROT_WRITE;
4750                        }
4751
4752                        hat_memload_array(hat, a, pgsz, ppa,
4753                            prot & vpprot, hat_flag);
4754
4755                        if (hat_flag & HAT_LOAD_LOCK) {
4756                                VM_STAT_ADD(segvnvmstats.fltanpages[7]);
4757                        } else {
4758                                VM_STAT_ADD(segvnvmstats.fltanpages[8]);
4759                                for (i = 0; i < pages; i++)
4760                                        page_unlock(ppa[i]);
4761                        }
4762                        if (vpage != NULL)
4763                                vpage += pages;
4764
4765                        anon_array_exit(&cookie);
4766                        adjszc_chk = 1;
4767                }
4768                if (a == lpgeaddr)
4769                        break;
4770                ASSERT(a < lpgeaddr);
4771                /*
4772                 * ierr == -1 means we failed to allocate a large page.
4773                 * so do a size down operation.
4774                 *
4775                 * ierr == -2 means some other process that privately shares
4776                 * pages with this process has allocated a larger page and we
4777                 * need to retry with larger pages. So do a size up
4778                 * operation. This relies on the fact that large pages are
4779                 * never partially shared i.e. if we share any constituent
4780                 * page of a large page with another process we must share the
4781                 * entire large page. Note this cannot happen for SOFTLOCK
4782                 * case, unless current address (a) is at the beginning of the
4783                 * next page size boundary because the other process couldn't
4784                 * have relocated locked pages.
4785                 */
4786                ASSERT(ierr == -1 || ierr == -2);
4787
4788                if (segvn_anypgsz) {
4789                        ASSERT(ierr == -2 || szc != 0);
4790                        ASSERT(ierr == -1 || szc < seg->s_szc);
4791                        szc = (ierr == -1) ? szc - 1 : szc + 1;
4792                } else {
4793                        /*
4794                         * For non COW faults and segvn_anypgsz == 0
4795                         * we need to be careful not to loop forever
4796                         * if existing page is found with szc other
4797                         * than 0 or seg->s_szc. This could be due
4798                         * to page relocations on behalf of DR or
4799                         * more likely large page creation. For this
4800                         * case simply re-size to existing page's szc
4801                         * if returned by anon_map_getpages().
4802                         */
4803                        if (ppa_szc == (uint_t)-1) {
4804                                szc = (ierr == -1) ? 0 : seg->s_szc;
4805                        } else {
4806                                ASSERT(ppa_szc <= seg->s_szc);
4807                                ASSERT(ierr == -2 || ppa_szc < szc);
4808                                ASSERT(ierr == -1 || ppa_szc > szc);
4809                                szc = ppa_szc;
4810                        }
4811                }
```



```
4813                pgsz = page_get_pagesize(szc);
4814                pages = btop(pgsz);
4815                ASSERT(type != F_SOFTLOCK || ierr == -1 ||
4816                    (IS_P2ALIGNED(a, pgsz) && IS_P2ALIGNED(lpgeaddr, pgsz)));
4817                if (type == F_SOFTLOCK) {
4818                        /*
4819                         * For softlocks we cannot reduce the fault area
4820                         * (calculated based on the largest page size for this
4821                         * segment) for size down and a is already next
4822                         * page size aligned as assertted above for size
4823                         * ups. Therefore just continue in case of softlock.
4824                         */
4825                        VM_STAT_ADD(segvnvmstats.fltanpages[9]);
4826                        continue; /* keep lint happy */
4827                } else if (ierr == -2) {
4828
4829                        /*
4830                         * Size up case. Note lpgaddr may only be needed for
4831                         * softlock case so we don't adjust it here.
4832                         */
4833                        VM_STAT_ADD(segvnvmstats.fltanpages[10]);
4834                        a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
4835                        ASSERT(a >= lpgaddr);
4836                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4837                        aindx = svd->anon_index + seg_page(seg, a);
4838                        vpage = (svd->vpage != NULL) ?
4839                            &svd->vpage[seg_page(seg, a)] : NULL;
4840                } else {
4841                        /*
4842                         * Size down case. Note lpgaddr may only be needed for
4843                         * softlock case so we don't adjust it here.
4844                         */
4845                        VM_STAT_ADD(segvnvmstats.fltanpages[11]);
4846                        ASSERT(IS_P2ALIGNED(a, pgsz));
4847                        ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
4848                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4849                        ASSERT(a < lpgeaddr);
4850                        if (a < addr) {
4851                                /*
4852                                 * The beginning of the large page region can
4853                                 * be pulled to the right to make a smaller
4854                                 * region. We haven't yet faulted a single
4855                                 * page.
4856                                 */
4857                                VM_STAT_ADD(segvnvmstats.fltanpages[12]);
4858                                a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
4859                                ASSERT(a >= lpgaddr);
4860                                aindx = svd->anon_index + seg_page(seg, a);
4861                                vpage = (svd->vpage != NULL) ?
4862                                    &svd->vpage[seg_page(seg, a)] : NULL;
4863                        }
4864                }
4865        }
4866        VM_STAT_ADD(segvnvmstats.fltanpages[13]);
4867        ANON_LOCK_EXIT(&amp->a_rwlock);
4868        kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
4869        return (0);
4870 error:
4871        VM_STAT_ADD(segvnvmstats.fltanpages[14]);
4872        ANON_LOCK_EXIT(&amp->a_rwlock);
4873        kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
4874        if (type == F_SOFTLOCK && a > lpgaddr) {
4875                VM_STAT_ADD(segvnvmstats.fltanpages[15]);
4876                segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4877        }
4878        return (err);
```

```
4879 }

4881 int fltadvice = 1;      /* set to free behind pages for sequential access */

4883 /*
4884  * This routine is called via a machine specific fault handling routine.
4885  * It is also called by software routines wishing to lock or unlock
4886  * a range of addresses.
4887  *
4888  * Here is the basic algorithm:
4889  *      If unlocking
4890  *              Call segvn_softunlock
4891  *              Return
4892  *      endif
4893  *      Checking and set up work
4894  *      If we will need some non-anonymous pages
4895  *              Call VOP_GETPAGE over the range of non-anonymous pages
4896  *      endif
4897  *      Loop over all addresses requested
4898  *              Call segvn_faultpage passing in page list
4899  *                  to load up translations and handle anonymous pages
4900  *      endloop
4901  *      Load up translation to any additional pages in page list not
4902  *          already handled that fit into this segment
4903  */
4904 static faultcode_t
4905 segvn_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
4906     enum fault_type type, enum seg_rw rw)
4907 {
4908         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4909         page_t **plp, **ppp, *pp;
4910         u_offset_t off;
4911         caddr_t a;
4912         struct vpage *vpage;
4913         uint_t vpprot, prot;
4914         int err;
4915         page_t *pl[PVN_GETPAGE_NUM + 1];
4916         size_t plsz, pl_alloc_sz;
4917         size_t page;
4918         ulong_t anon_index;
4919         struct anon_map *amp;
4920         int dogetpage = 0;
4921         caddr_t lpgaddr, lpgeaddr;
4922         size_t pgsz;
4923         anon_sync_obj_t cookie;
4924         int brkcow = BREAK_COW_SHARE(rw, type, svd->type);

4926         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
4927         ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);

4929         /*
4930          * First handle the easy stuff
4931          */
4932         if (type == F_SOFTUNLOCK) {
4933                 if (rw == S_READ_NOCOW) {
4934                         rw = S_READ;
4935                         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
4936                 }
4937                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4938                 pgsz = (seg->s_szc == 0) ? PAGESIZE :
4939                     page_get_pagesize(seg->s_szc);
4940                 VM_STAT_COND_ADD(pgsz > PAGESIZE, segvnvmstats.fltanpages[16]);
4941                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
4942                 segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
4943                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4944                 return (0);
```



```
4945         }

4947         ASSERT(svd->tr_state == SEGVN_TR_OFF ||
4948             !HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
4949         if (brkcow == 0) {
4950                 if (svd->tr_state == SEGVN_TR_INIT) {
4951                         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4952                         if (svd->tr_state == SEGVN_TR_INIT) {
4953                                 ASSERT(svd->vp != NULL && svd->amp == NULL);
4954                                 ASSERT(svd->flags & MAP_TEXT);
4955                                 ASSERT(svd->type == MAP_PRIVATE);
4956                                 segvn_textrepl(seg);
4957                                 ASSERT(svd->tr_state != SEGVN_TR_INIT);
4958                                 ASSERT(svd->tr_state != SEGVN_TR_ON ||
4959                                     svd->amp != NULL);
4960                         }
4961                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4962                 }
4963         } else if (svd->tr_state != SEGVN_TR_OFF) {
4964                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

4966                 if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
4967                         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
4968                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4969                         return (FC_PROT);
4970                 }

4972                 if (svd->tr_state == SEGVN_TR_ON) {
4973                         ASSERT(svd->vp != NULL && svd->amp != NULL);
4974                         segvn_textunrepl(seg, 0);
4975                         ASSERT(svd->amp == NULL &&
4976                             svd->tr_state == SEGVN_TR_OFF);
4977                 } else if (svd->tr_state != SEGVN_TR_OFF) {
4978                         svd->tr_state = SEGVN_TR_OFF;
4979                 }
4980                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
4981                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4982         }

4984 top:
4985         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

4987         /*
4988          * If we have the same protections for the entire segment,
4989          * insure that the access being attempted is legitimate.
4990          */

4992         if (svd->pageprot == 0) {
4993                 uint_t protchk;

4995                 switch (rw) {
4996                 case S_READ:
4997                 case S_READ_NOCOW:
4998                         protchk = PROT_READ;
4999                         break;
5000                 case S_WRITE:
5001                         protchk = PROT_WRITE;
5002                         break;
5003                 case S_EXEC:
5004                         protchk = PROT_EXEC;
5005                         break;
5006                 case S_OTHER:
5007                 default:
5008                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
5009                         break;
5010                 }
```

```
5012                if ((svd->prot & protchk) == 0) {
5013                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5014                        return (FC_PROT);       /* illegal access type */
5015                }
5016        }

5018        if (brkcow && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5019                /* this must be SOFTLOCK S_READ fault */
5020                ASSERT(svd->amp == NULL);
5021                ASSERT(svd->tr_state == SEGVN_TR_OFF);
5022                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5023                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5024                if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5025                        /*
5026                         * this must be the first ever non S_READ_NOCOW
5027                         * softlock for this segment.
5028                         */
5029                        ASSERT(svd->softlockcnt == 0);
5030                        hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5031                            HAT_REGION_TEXT);
5032                        svd->rcookie = HAT_INVALID_REGION_COOKIE;
5033                }
5034                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5035                goto top;
5036        }

5038        /*
5039         * We can't allow the long term use of softlocks for vmpss segments,
5040         * because in some file truncation cases we should be able to demote
5041         * the segment, which requires that there are no softlocks.  The
5042         * only case where it's ok to allow a SOFTLOCK fault against a vmpss
5043         * segment is S_READ_NOCOW, where the caller holds the address space
5044         * locked as writer and calls softunlock before dropping the as lock.
5045         * S_READ_NOCOW is used by /proc to read memory from another user.
5046         *
5047         * Another deadlock between SOFTLOCK and file truncation can happen
5048         * because segvn_fault_vnodepages() calls the FS one pagesize at
5049         * a time. A second VOP_GETPAGE() call by segvn_fault_vnodepages()
5050         * can cause a deadlock because the first set of page_t's remain
5051         * locked SE_SHARED.  To avoid this, we demote segments on a first
5052         * SOFTLOCK if they have a length greater than the segment's
5053         * page size.
5054         *
5055         * So for now, we only avoid demoting a segment on a SOFTLOCK when
5056         * the access type is S_READ_NOCOW and the fault length is less than
5057         * or equal to the segment's page size. While this is quite restrictive,
5058         * it should be the most common case of SOFTLOCK against a vmpss
5059         * segment.
5060         *
5061         * For S_READ_NOCOW, it's safe not to do a copy on write because the
5062         * caller makes sure no COW will be caused by another thread for a
5063         * softlocked page.
5064         */
5065        if (type == F_SOFTLOCK && svd->vp != NULL && seg->s_szc != 0) {
5066                int demote = 0;

5068                if (rw != S_READ_NOCOW) {
5069                        demote = 1;
5070                }
5071                if (!demote && len > PAGESIZE) {
5072                        pgsz = page_get_pagesize(seg->s_szc);
5073                        CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
5074                            lpgeaddr);
5075                        if (lpgeaddr - lpgaddr > pgsz) {
5076                                demote = 1;
```



```
5077                        }
5078                }

5080                ASSERT(demote || AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

5082                if (demote) {
5083                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5084                        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5085                        if (seg->s_szc != 0) {
5086                                segvn_vmpss_clrszc_cnt++;
5087                                ASSERT(svd->softlockcnt == 0);
5088                                err = segvn_clrszc(seg);
5089                                if (err) {
5090                                        segvn_vmpss_clrszc_err++;
5091                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5092                                        return (FC_MAKE_ERR(err));
5093                                }
5094                        }
5095                        ASSERT(seg->s_szc == 0);
5096                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5097                        goto top;
5098                }
5099        }

5101        /*
5102         * Check to see if we need to allocate an anon_map structure.
5103         */
5104        if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
5105                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5106                /*
5107                 * Drop the "read" lock on the segment and acquire
5108                 * the "write" version since we have to allocate the
5109                 * anon_map.
5110                 */
5111                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5112                SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

5114                if (svd->amp == NULL) {
5115                        svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
5116                        svd->amp->a_szc = seg->s_szc;
5117                }
5118                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

5120                /*
5121                 * Start all over again since segment protections
5122                 * may have changed after we dropped the "read" lock.
5123                 */
5124                goto top;
5125        }

5127        /*
5128         * S_READ_NOCOW vs S_READ distinction was
5129         * only needed for the code above. After
5130         * that we treat it as S_READ.
5131         */
5132        if (rw == S_READ_NOCOW) {
5133                ASSERT(type == F_SOFTLOCK);
5134                ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
5135                rw = S_READ;
5136        }

5138        amp = svd->amp;

5140        /*
5141         * MADV_SEQUENTIAL work is ignored for large page segments.
5142         */
```

```
5143         if (seg->s_szc != 0) {
5144                 pgsz = page_get_pagesize(seg->s_szc);
5145                 ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
5146                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
5147                 if (svd->vp == NULL) {
5148                         err = segvn_fault_anonpages(hat, seg, lpgaddr,
5149                             lpgeaddr, type, rw, addr, addr + len, brkcow);
5150                 } else {
5151                         err = segvn_fault_vnodepages(hat, seg, lpgaddr,
5152                             lpgeaddr, type, rw, addr, addr + len, brkcow);
5153                         if (err == IE_RETRY) {
5154                                 ASSERT(seg->s_szc == 0);
5155                                 ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
5156                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5157                                 goto top;
5158                         }
5159                 }
5160                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5161                 return (err);
5162         }

5164         page = seg_page(seg, addr);
5165         if (amp != NULL) {
5166                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5167                 anon_index = svd->anon_index + page;

5169                 if (type == F_PROT && rw == S_READ &&
5170                     svd->tr_state == SEGVN_TR_OFF &&
5171                     svd->type == MAP_PRIVATE && svd->pageprot == 0) {
5172                         size_t index = anon_index;
5173                         struct anon *ap;

5175                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5176                         /*
5177                          * The fast path could apply to S_WRITE also, except
5178                          * that the protection fault could be caused by lazy
5179                          * tlb flush when ro->rw. In this case, the pte is
5180                          * RW already. But RO in the other cpu's tlb causes
5181                          * the fault. Since hat_chgprot won't do anything if
5182                          * pte doesn't change, we may end up faulting
5183                          * indefinitely until the RO tlb entry gets replaced.
5184                          */
5185                         for (a = addr; a < addr + len; a += PAGESIZE, index++) {
5186                                 anon_array_enter(amp, index, &cookie);
5187                                 ap = anon_get_ptr(amp->ahp, index);
5188                                 anon_array_exit(&cookie);
5189                                 if ((ap == NULL) || (ap->an_refcnt != 1)) {
5190                                         ANON_LOCK_EXIT(&amp->a_rwlock);
5191                                         goto slow;
5192                                 }
5193                         }
5194                         hat_chgprot(seg->s_as->a_hat, addr, len, svd->prot);
5195                         ANON_LOCK_EXIT(&amp->a_rwlock);
5196                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5197                         return (0);
5198                 }
5199         }
5200 slow:

5202         if (svd->vpage == NULL)
5203                 vpage = NULL;
5204         else
5205                 vpage = &svd->vpage[page];

5207         off = svd->offset + (uintptr_t)(addr - seg->s_base);
```



```
5209         /*
5210          * If MADV_SEQUENTIAL has been set for the particular page we
5211          * are faulting on, free behind all pages in the segment and put
5212          * them on the free list.
5213          */

5215         if ((page != 0) && fltadvice && svd->tr_state != SEGVN_TR_ON) {
5216                 struct vpage *vpp;
5217                 ulong_t fanon_index;
5218                 size_t fpage;
5219                 u_offset_t pgoff, fpgoff;
5220                 struct vnode *fvp;
5221                 struct anon *fap = NULL;

5223                 if (svd->advice == MADV_SEQUENTIAL ||
5224                     (svd->pageadvice &&
5225                     VPP_ADVICE(vpage) == MADV_SEQUENTIAL)) {
5226                         pgoff = off - PAGESIZE;
5227                         fpage = page - 1;
5228                         if (vpage != NULL)
5229                                 vpp = &svd->vpage[fpage];
5230                         if (amp != NULL)
5231                                 fanon_index = svd->anon_index + fpage;

5233                         while (pgoff > svd->offset) {
5234                                 if (svd->advice != MADV_SEQUENTIAL &&
5235                                     (!svd->pageadvice || (vpage &&
5236                                     VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
5237                                         break;

5239                                 /*
5240                                  * If this is an anon page, we must find the
5241                                  * correct <vp, offset> for it
5242                                  */
5243                                 fap = NULL;
5244                                 if (amp != NULL) {
5245                                         ANON_LOCK_ENTER(&amp->a_rwlock,
5246                                             RW_READER);
5247                                         anon_array_enter(amp, fanon_index,
5248                                             &cookie);
5249                                         fap = anon_get_ptr(amp->ahp,
5250                                             fanon_index);
5251                                         if (fap != NULL) {
5252                                                 swap_xlate(fap, &fvp, &fpgoff);
5253                                         } else {
5254                                                 fpgoff = pgoff;
5255                                                 fvp = svd->vp;
5256                                         }
5257                                         anon_array_exit(&cookie);
5258                                         ANON_LOCK_EXIT(&amp->a_rwlock);
5259                                 } else {
5260                                         fpgoff = pgoff;
5261                                         fvp = svd->vp;
5262                                 }
5263                                 if (fvp == NULL)
5264                                         break;  /* XXX */
5265                                 /*
5266                                  * Skip pages that are free or have an
5267                                  * "exclusive" lock.
5268                                  */
5269                                 pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
5270                                 if (pp == NULL)
5271                                         break;
5272                                 /*
5273                                  * We don't need the page_struct_lock to test
5274                                  * as this is only advisory; even if we
```

```
5275                                 * acquire it someone might race in and lock
5276                                 * the page after we unlock and before the
5277                                 * PUTPAGE, then VOP_PUTPAGE will do nothing.
5278                                 */
5279                                if (pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
5280                                        /*
5281                                         * Hold the vnode before releasing
5282                                         * the page lock to prevent it from
5283                                         * being freed and re-used by some
5284                                         * other thread.
5285                                         */
5286                                        VN_HOLD(fvp);
5287                                        page_unlock(pp);
5288                                        /*
5289                                         * We should build a page list
5290                                         * to kluster putpages XXX
5291                                         */
5292                                        (void) VOP_PUTPAGE(fvp,
5293                                            (offset_t)fpgoff, PAGESIZE,
5294                                            (B_DONTNEED|B_FREE|B_ASYNC),
5295                                            svd->cred, NULL);
5296                                        VN_RELE(fvp);
5297                                } else {
5298                                        /*
5299                                         * XXX - Should the loop terminate if
5300                                         * the page is 'locked'?
5301                                         */
5302                                        page_unlock(pp);
5303                                }
5304                                --vpp;
5305                                --fanon_index;
5306                                pgoff -= PAGESIZE;
5307                        }
5308                }
5309        }

5311        plp = pl;
5312        *plp = NULL;
5313        pl_alloc_sz = 0;

5315        /*
5316         * See if we need to call VOP_GETPAGE for
5317         * *any* of the range being faulted on.
5318         * We can skip all of this work if there
5319         * was no original vnode.
5320         */
5321        if (svd->vp != NULL) {
5322                u_offset_t vp_off;
5323                size_t vp_len;
5324                struct anon *ap;
5325                vnode_t *vp;

5327                vp_off = off;
5328                vp_len = len;

5330                if (amp == NULL)
5331                        dogetpage = 1;
5332                else {
5333                        /*
5334                         * Only acquire reader lock to prevent amp->ahp
5335                         * from being changed.  It's ok to miss pages,
5336                         * hence we don't do anon_array_enter
5337                         */
5338                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5339                        ap = anon_get_ptr(amp->ahp, anon_index);
```



```
5341                        if (len <= PAGESIZE)
5342                                /* inline non_anon() */
5343                                dogetpage = (ap == NULL);
5344                        else
5345                                dogetpage = non_anon(amp->ahp, anon_index,
5346                                    &vp_off, &vp_len);
5347                        ANON_LOCK_EXIT(&amp->a_rwlock);
5348                }

5350                if (dogetpage) {
5351                        enum seg_rw arw;
5352                        struct as *as = seg->s_as;

5354                        if (len > ptob((sizeof (pl) / sizeof (pl[0])) - 1)) {
5355                                /*
5356                                 * Page list won't fit in local array,
5357                                 * allocate one of the needed size.
5358                                 */
5359                                pl_alloc_sz =
5360                                    (btop(len) + 1) * sizeof (page_t *);
5361                                plp = kmem_alloc(pl_alloc_sz, KM_SLEEP);
5362                                plp[0] = NULL;
5363                                plsz = len;
5364                        } else if (rw == S_WRITE && svd->type == MAP_PRIVATE ||
5365                            svd->tr_state == SEGVN_TR_ON || rw == S_OTHER ||
5366                            (((size_t)(addr + PAGESIZE) <
5367                            (size_t)(seg->s_base + seg->s_size)) &&
5368                            hat_probe(as->a_hat, addr + PAGESIZE))) {
5369                                /*
5370                                 * Ask VOP_GETPAGE to return the exact number
5371                                 * of pages if
5372                                 * (a) this is a COW fault, or
5373                                 * (b) this is a software fault, or
5374                                 * (c) next page is already mapped.
5375                                 */
5376                                plsz = len;
5377                        } else {
5378                                /*
5379                                 * Ask VOP_GETPAGE to return adjacent pages
5380                                 * within the segment.
5381                                 */
5382                                plsz = MIN((size_t)PVN_GETPAGE_SZ, (size_t)
5383                                    ((seg->s_base + seg->s_size) - addr));
5384                                ASSERT((addr + plsz) <=
5385                                    (seg->s_base + seg->s_size));
5386                        }

5388                        /*
5389                         * Need to get some non-anonymous pages.
5390                         * We need to make only one call to GETPAGE to do
5391                         * this to prevent certain deadlocking conditions
5392                         * when we are doing locking.  In this case
5393                         * non_anon() should have picked up the smallest
5394                         * range which includes all the non-anonymous
5395                         * pages in the requested range.  We have to
5396                         * be careful regarding which rw flag to pass in
5397                         * because on a private mapping, the underlying
5398                         * object is never allowed to be written.
5399                         */
5400                        if (rw == S_WRITE && svd->type == MAP_PRIVATE) {
5401                                arw = S_READ;
5402                        } else {
5403                                arw = rw;
5404                        }
5405                        vp = svd->vp;
5406                        TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
```

```
5407                            "segvn_getpage:seg %p addr %p vp %p",
5408                            seg, addr, vp);
5409                        err = VOP_GETPAGE(vp, (offset_t)vp_off, vp_len,
5410                            &vpprot, plp, plsz, seg, addr + (vp_off - off), arw,
5411                            svd->cred, NULL);
5412                        if (err) {
5413                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5414                                segvn_pagelist_rele(plp);
5415                                if (pl_alloc_sz)
5416                                        kmem_free(plp, pl_alloc_sz);
5417                                return (FC_MAKE_ERR(err));
5418                        }
5419                        if (svd->type == MAP_PRIVATE)
5420                                vpprot &= ~PROT_WRITE;
5421                }
5422        }

5424         /*
5425          * N.B. at this time the plp array has all the needed non-anon
5426          * pages in addition to (possibly) having some adjacent pages.
5427          */

5429         /*
5430          * Always acquire the anon_array_lock to prevent
5431          * 2 threads from allocating separate anon slots for
5432          * the same "addr".
5433          *
5434          * If this is a copy-on-write fault and we don't already
5435          * have the anon_array_lock, acquire it to prevent the
5436          * fault routine from handling multiple copy-on-write faults
5437          * on the same "addr" in the same address space.
5438          *
5439          * Only one thread should deal with the fault since after
5440          * it is handled, the other threads can acquire a translation
5441          * to the newly created private page.  This prevents two or
5442          * more threads from creating different private pages for the
5443          * same fault.
5444          *
5445          * We grab "serialization" lock here if this is a MAP_PRIVATE segment
5446          * to prevent deadlock between this thread and another thread
5447          * which has soft-locked this page and wants to acquire serial_lock.
5448          * ( bug 4026339 )
5449          *
5450          * The fix for bug 4026339 becomes unnecessary when using the
5451          * locking scheme with per amp rwlock and a global set of hash
5452          * lock, anon_array_lock.  If we steal a vnode page when low
5453          * on memory and upgrad the page lock through page_rename,
5454          * then the page is PAGE_HANDLED, nothing needs to be done
5455          * for this page after returning from segvn_faultpage.
5456          *
5457          * But really, the page lock should be downgraded after
5458          * the stolen page is page_rename'd.
5459          */

5461         if (amp != NULL)
5462                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

5464         /*
5465          * Ok, now loop over the address range and handle faults
5466          */
5467         for (a = addr; a < addr + len; a += PAGESIZE, off += PAGESIZE) {
5468                 err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
5469                     type, rw, brkcow);
5470                 if (err) {
5471                         if (amp != NULL)
5472                                 ANON_LOCK_EXIT(&amp->a_rwlock);
```



```
5473                         if (type == F_SOFTLOCK && a > addr) {
5474                                 segvn_softunlock(seg, addr, (a - addr),
5475                                     S_OTHER);
5476                         }
5477                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5478                         segvn_pagelist_rele(plp);
5479                         if (pl_alloc_sz)
5480                                 kmem_free(plp, pl_alloc_sz);
5481                         return (err);
5482                 }
5483                 if (vpage) {
5484                         vpage++;
5485                 } else if (svd->vpage) {
5486                         page = seg_page(seg, addr);
5487                         vpage = &svd->vpage[++page];
5488                 }
5489         }

5491         /* Didn't get pages from the underlying fs so we're done */
5492         if (!dogetpage)
5493                 goto done;

5495         /*
5496          * Now handle any other pages in the list returned.
5497          * If the page can be used, load up the translations now.
5498          * Note that the for loop will only be entered if "plp"
5499          * is pointing to a non-NULL page pointer which means that
5500          * VOP_GETPAGE() was called and vpprot has been initialized.
5501          */
5502         if (svd->pageprot == 0)
5503                 prot = svd->prot & vpprot;


5506         /*
5507          * Large Files: diff should be unsigned value because we started
5508          * supporting > 2GB segment sizes from 2.5.1 and when a
5509          * large file of size > 2GB gets mapped to address space
5510          * the diff value can be > 2GB.
5511          */

5513         for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
5514                 size_t diff;
5515                 struct anon *ap;
5516                 int anon_index;
5517                 anon_sync_obj_t cookie;
5518                 int hat_flag = HAT_LOAD_ADV;

5520                 if (svd->flags & MAP_TEXT) {
5521                         hat_flag |= HAT_LOAD_TEXT;
5522                 }

5524                 if (pp == PAGE_HANDLED)
5525                         continue;

5527                 if (svd->tr_state != SEGVN_TR_ON &&
5528                     pp->p_offset >=  svd->offset &&
5529                     pp->p_offset < svd->offset + seg->s_size) {

5531                         diff = pp->p_offset - svd->offset;

5533                         /*
5534                          * Large Files: Following is the assertion
5535                          * validating the above cast.
5536                          */
5537                         ASSERT(svd->vp == pp->p_vnode);
```

```
5539                        page = btop(diff);
5540                        if (svd->pageprot)
5541                                prot = VPP_PROT(&svd->vpage[page]) & vpprot;
5542
5543                        /*
5544                         * Prevent other threads in the address space from
5545                         * creating private pages (i.e., allocating anon slots)
5546                         * while we are in the process of loading translations
5547                         * to additional pages returned by the underlying
5548                         * object.
5549                         */
5550                        if (amp != NULL) {
5551                                anon_index = svd->anon_index + page;
5552                                anon_array_enter(amp, anon_index, &cookie);
5553                                ap = anon_get_ptr(amp->ahp, anon_index);
5554                        }
5555                        if ((amp == NULL) || (ap == NULL)) {
5556                                if (IS_VMODSORT(pp->p_vnode) ||
5557                                    enable_mbit_wa) {
5558                                        if (rw == S_WRITE)
5559                                                hat_setmod(pp);
5560                                        else if (rw != S_OTHER &&
5561                                            !hat_ismod(pp))
5562                                                prot &= ~PROT_WRITE;
5563                                }
5564                                /*
5565                                 * Skip mapping read ahead pages marked
5566                                 * for migration, so they will get migrated
5567                                 * properly on fault
5568                                 */
5569                                ASSERT(amp == NULL ||
5570                                    svd->rcookie == HAT_INVALID_REGION_COOKIE);
5571                                if ((prot & PROT_READ) && !PP_ISMIGRATE(pp)) {
5572                                        hat_memload_region(hat,
5573                                            seg->s_base + diff,
5574                                            pp, prot, hat_flag,
5575                                            svd->rcookie);
5576                                }
5577                        }
5578                        if (amp != NULL)
5579                                anon_array_exit(&cookie);
5580                }
5581                page_unlock(pp);
5582        }
5583 done:
5584        if (amp != NULL)
5585                ANON_LOCK_EXIT(&amp->a_rwlock);
5586        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5587        if (pl_alloc_sz)
5588                kmem_free(plp, pl_alloc_sz);
5589        return (0);
5590 }
5591
5592 /*
5593  * This routine is used to start I/O on pages asynchronously.  XXX it will
5594  * only create PAGESIZE pages. At fault time they will be relocated into
5595  * larger pages.
5596  */
5597 static faultcode_t
5598 segvn_faulta(struct seg *seg, caddr_t addr)
5599 {
5600        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
5601        int err;
5602        struct anon_map *amp;
5603        vnode_t *vp;
```



```
5605        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
5606
5607        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
5608        if ((amp = svd->amp) != NULL) {
5609                struct anon *ap;
5610
5611                /*
5612                 * Reader lock to prevent amp->ahp from being changed.
5613                 * This is advisory, it's ok to miss a page, so
5614                 * we don't do anon_array_enter lock.
5615                 */
5616                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5617                if ((ap = anon_get_ptr(amp->ahp,
5618                    svd->anon_index + seg_page(seg, addr))) != NULL) {
5619
5620                        err = anon_getpage(&ap, NULL, NULL,
5621                            0, seg, addr, S_READ, svd->cred);
5622
5623                        ANON_LOCK_EXIT(&amp->a_rwlock);
5624                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5625                        if (err)
5626                                return (FC_MAKE_ERR(err));
5627                        return (0);
5628                }
5629                ANON_LOCK_EXIT(&amp->a_rwlock);
5630        }
5631
5632        if (svd->vp == NULL) {
5633                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5634                return (0);                     /* zfod page - do nothing now */
5635        }
5636
5637        vp = svd->vp;
5638        TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
5639            "segvn_getpage:seg %p addr %p vp %p", seg, addr, vp);
5640        err = VOP_GETPAGE(vp,
5641            (offset_t)(svd->offset + (uintptr_t)(addr - seg->s_base)),
5642            PAGESIZE, NULL, NULL, 0, seg, addr,
5643            S_OTHER, svd->cred, NULL);
5644
5645        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5646        if (err)
5647                return (FC_MAKE_ERR(err));
5648        return (0);
5649 }
5650
5651 static int
5652 segvn_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
5653 {
5654        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
5655        struct vpage *cvp, *svp, *evp;
5656        struct vnode *vp;
5657        size_t pgsz;
5658        pgcnt_t pgcnt;
5659        anon_sync_obj_t cookie;
5660        int unload_done = 0;
5661
5662        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
5663
5664        if ((svd->maxprot & prot) != prot)
5665                return (EACCES);                        /* violated maxprot */
5666
5667        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5668
5669        /* return if prot is the same */
5670        if (!svd->pageprot && svd->prot == prot) {
```

```
5671                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5672                return (0);
5673        }

5675        /*
5676         * Since we change protections we first have to flush the cache.
5677         * This makes sure all the pagelock calls have to recheck
5678         * protections.
5679         */
5680        if (svd->softlockcnt > 0) {
5681                ASSERT(svd->tr_state == SEGVN_TR_OFF);

5683                /*
5684                 * If this is shared segment non 0 softlockcnt
5685                 * means locked pages are still in use.
5686                 */
5687                if (svd->type == MAP_SHARED) {
5688                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5689                        return (EAGAIN);
5690                }

5692                /*
5693                 * Since we do have the segvn writers lock nobody can fill
5694                 * the cache with entries belonging to this seg during
5695                 * the purge. The flush either succeeds or we still have
5696                 * pending I/Os.
5697                 */
5698                segvn_purge(seg);
5699                if (svd->softlockcnt > 0) {
5700                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5701                        return (EAGAIN);
5702                }
5703        }

5705        if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5706                ASSERT(svd->amp == NULL);
5707                ASSERT(svd->tr_state == SEGVN_TR_OFF);
5708                hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5709                    HAT_REGION_TEXT);
5710                svd->rcookie = HAT_INVALID_REGION_COOKIE;
5711                unload_done = 1;
5712        } else if (svd->tr_state == SEGVN_TR_INIT) {
5713                svd->tr_state = SEGVN_TR_OFF;
5714        } else if (svd->tr_state == SEGVN_TR_ON) {
5715                ASSERT(svd->amp != NULL);
5716                segvn_textunrepl(seg, 0);
5717                ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
5718                unload_done = 1;
5719        }

5721        if ((prot & PROT_WRITE) && svd->type == MAP_SHARED &&
5722            svd->vp != NULL && (svd->vp->v_flag & VVMEXEC)) {
5723                ASSERT(vn_is_mapped(svd->vp, V_WRITE));
5724                segvn_inval_trcache(svd->vp);
5725        }
5726        if (seg->s_szc != 0) {
5727                int err;
5728                pgsz = page_get_pagesize(seg->s_szc);
5729                pgcnt = pgsz >> PAGESHIFT;
5730                ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
5731                if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
5732                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5733                        ASSERT(seg->s_base != addr || seg->s_size != len);
5734                        /*
5735                         * If we are holding the as lock as a reader then
5736                         * we need to return IE_RETRY and let the as
```



```
5737                         * layer drop and re-acquire the lock as a writer.
5738                         */
5739                        if (AS_READ_HELD(seg->s_as, &seg->s_as->a_lock))
5740                                return (IE_RETRY);
5741                        VM_STAT_ADD(segvnvmstats.demoterange[1]);
5742                        if (svd->type == MAP_PRIVATE || svd->vp != NULL) {
5743                                err = segvn_demote_range(seg, addr, len,
5744                                    SDR_END, 0);
5745                        } else {
5746                                uint_t szcvec = map_pgszcvec(seg->s_base,
5747                                    pgsz, (uintptr_t)seg->s_base,
5748                                    (svd->flags & MAP_TEXT), MAPPGSZC_SHM, 0);
5749                                err = segvn_demote_range(seg, addr, len,
5750                                    SDR_END, szcvec);
5751                        }
5752                        if (err == 0)
5753                                return (IE_RETRY);
5754                        if (err == ENOMEM)
5755                                return (IE_NOMEM);
5756                        return (err);
5757                }
5758        }


5761        /*
5762         * If it's a private mapping and we're making it writable then we
5763         * may have to reserve the additional swap space now. If we are
5764         * making writable only a part of the segment then we use its vpage
5765         * array to keep a record of the pages for which we have reserved
5766         * swap. In this case we set the pageswap field in the segment's
5767         * segvn structure to record this.
5768         *
5769         * If it's a private mapping to a file (i.e., vp != NULL) and we're
5770         * removing write permission on the entire segment and we haven't
5771         * modified any pages, we can release the swap space.
5772         */
5773        if (svd->type == MAP_PRIVATE) {
5774                if (prot & PROT_WRITE) {
5775                        if (!(svd->flags & MAP_NORESERVE) &&
5776                            !(svd->swresv && svd->pageswap == 0)) {
5777                                size_t sz = 0;

5779                                /*
5780                                 * Start by determining how much swap
5781                                 * space is required.
5782                                 */
5783                                if (addr == seg->s_base &&
5784                                    len == seg->s_size &&
5785                                    svd->pageswap == 0) {
5786                                        /* The whole segment */
5787                                        sz = seg->s_size;
5788                                } else {
5789                                        /*
5790                                         * Make sure that the vpage array
5791                                         * exists, and make a note of the
5792                                         * range of elements corresponding
5793                                         * to len.
5794                                         */
5795                                        segvn_vpage(seg);
5796                                        svp = &svd->vpage[seg_page(seg, addr)];
5797                                        evp = &svd->vpage[seg_page(seg,
5798                                            addr + len)];

5800                                        if (svd->pageswap == 0) {
5801                                                /*
5802                                                 * This is the first time we've
```

```
5803                                             * asked for a part of this
5804                                             * segment, so we need to
5805                                             * reserve everything we've
5806                                             * been asked for.
5807                                             */
5808                                            sz = len;
5809                                    } else {
5810                                            /*
5811                                             * We have to count the number
5812                                             * of pages required.
5813                                             */
5814                                            for (cvp = svp;  cvp < evp;
5815                                                cvp++) {
5816                                                    if (!VPP_ISSWAPRES(cvp))
5817                                                            sz++;
5818                                            }
5819                                            sz <<= PAGESHIFT;
5820                                    }
5821                            }

5823                            /* Try to reserve the necessary swap. */
5824                            if (anon_resv_zone(sz,
5825                                seg->s_as->a_proc->p_zone) == 0) {
5826                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5827                                return (IE_NOMEM);
5828                            }

5830                            /*
5831                             * Make a note of how much swap space
5832                             * we've reserved.
5833                             */
5834                            if (svd->pageswap == 0 && sz == seg->s_size) {
5835                                    svd->swresv = sz;
5836                            } else {
5837                                    ASSERT(svd->vpage != NULL);
5838                                    svd->swresv += sz;
5839                                    svd->pageswap = 1;
5840                                    for (cvp = svp; cvp < evp; cvp++) {
5841                                            if (!VPP_ISSWAPRES(cvp))
5842                                                    VPP_SETSWAPRES(cvp);
5843                                    }
5844                            }
5845                    }
5846            } else {
5847                    /*
5848                     * Swap space is released only if this segment
5849                     * does not map anonymous memory, since read faults
5850                     * on such segments still need an anon slot to read
5851                     * in the data.
5852                     */
5853                    if (svd->swresv != 0 && svd->vp != NULL &&
5854                        svd->amp == NULL && addr == seg->s_base &&
5855                        len == seg->s_size && svd->pageprot == 0) {
5856                            ASSERT(svd->pageswap == 0);
5857                            anon_unresv_zone(svd->swresv,
5858                                seg->s_as->a_proc->p_zone);
5859                            svd->swresv = 0;
5860                            TRACE_3(TR_FAC_VM, TR_ANON_PROC,
5861                                "anon proc:%p %lu %u", seg, 0, 0);
5862                    }
5863            }
5864    }

5866    if (addr == seg->s_base && len == seg->s_size && svd->vpage == NULL) {
5867            if (svd->prot == prot) {
5868                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
```



```
5869                            return (0);                     /* all done */
5870                    }
5871                    svd->prot = (uchar_t)prot;
5872            } else if (svd->type == MAP_PRIVATE) {
5873                    struct anon *ap = NULL;
5874                    page_t *pp;
5875                    u_offset_t offset, off;
5876                    struct anon_map *amp;
5877                    ulong_t anon_idx = 0;

5879                    /*
5880                     * A vpage structure exists or else the change does not
5881                     * involve the entire segment.  Establish a vpage structure
5882                     * if none is there.  Then, for each page in the range,
5883                     * adjust its individual permissions.  Note that write-
5884                     * enabling a MAP_PRIVATE page can affect the claims for
5885                     * locked down memory.  Overcommitting memory terminates
5886                     * the operation.
5887                     */
5888                    segvn_vpage(seg);
5889                    svd->pageprot = 1;
5890                    if ((amp = svd->amp) != NULL) {
5891                            anon_idx = svd->anon_index + seg_page(seg, addr);
5892                            ASSERT(seg->s_szc == 0 ||
5893                                IS_P2ALIGNED(anon_idx, pgcnt));
5894                            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5895                    }

5897                    offset = svd->offset + (uintptr_t)(addr - seg->s_base);
5898                    evp = &svd->vpage[seg_page(seg, addr + len)];

5900                    /*
5901                     * See Statement at the beginning of segvn_lockop regarding
5902                     * the way cowcnts and lckcnts are handled.
5903                     */
5904                    for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {

5906                            if (seg->s_szc != 0) {
5907                                    if (amp != NULL) {
5908                                            anon_array_enter(amp, anon_idx,
5909                                                &cookie);
5910                                    }
5911                                    if (IS_P2ALIGNED(anon_idx, pgcnt) &&
5912                                        !segvn_claim_pages(seg, svp, offset,
5913                                        anon_idx, prot)) {
5914                                            if (amp != NULL) {
5915                                                    anon_array_exit(&cookie);
5916                                            }
5917                                            break;
5918                                    }
5919                                    if (amp != NULL) {
5920                                            anon_array_exit(&cookie);
5921                                    }
5922                                    anon_idx++;
5923                            } else {
5924                                    if (amp != NULL) {
5925                                            anon_array_enter(amp, anon_idx,
5926                                                &cookie);
5927                                            ap = anon_get_ptr(amp->ahp, anon_idx++);
5928                                    }

5930                                    if (VPP_ISPPLOCK(svp) &&
5931                                        VPP_PROT(svp) != prot) {

5933                                            if (amp == NULL || ap == NULL) {
5934                                                    vp = svd->vp;
```

```
5935                                            off = offset;
5936                                    } else
5937                                            swap_xlate(ap, &vp, &off);
5938                                    if (amp != NULL)
5939                                            anon_array_exit(&cookie);
5941                                    if ((pp = page_lookup(vp, off,
5942                                        SE_SHARED)) == NULL) {
5943                                            panic("segvn_setprot: no page");
5944                                            /*NOTREACHED*/
5945                                    }
5946                                    ASSERT(seg->s_szc == 0);
5947                                    if ((VPP_PROT(svp) ^ prot) &
5948                                        PROT_WRITE) {
5949                                            if (prot & PROT_WRITE) {
5950                                                    if (!page_addclaim(
5951                                                        pp)) {
5952                                                            page_unlock(pp);
5953                                                            break;
5954                                                    }
5955                                            } else {
5956                                                    if (!page_subclaim(
5957                                                        pp)) {
5958                                                            page_unlock(pp);
5959                                                            break;
5960                                                    }
5961                                            }
5962                                    }
5963                                    page_unlock(pp);
5964                            } else if (amp != NULL)
5965                                    anon_array_exit(&cookie);
5966                    }
5967                    VPP_SETPROT(svp, prot);
5968                    offset += PAGESIZE;
5969            }
5970            if (amp != NULL)
5971                    ANON_LOCK_EXIT(&amp->a_rwlock);
5973            /*
5974             * Did we terminate prematurely?  If so, simply unload
5975             * the translations to the things we've updated so far.
5976             */
5977            if (svp != evp) {
5978                    if (unload_done) {
5979                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5980                            return (IE_NOMEM);
5981                    }
5982                    len = (svp - &svd->vpage[seg_page(seg, addr)]) *
5983                        PAGESIZE;
5984                    ASSERT(seg->s_szc == 0 || IS_P2ALIGNED(len, pgsz));
5985                    if (len != 0)
5986                            hat_unload(seg->s_as->a_hat, addr,
5987                                len, HAT_UNLOAD);
5988                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5989                    return (IE_NOMEM);
5990            }
5991    } else {
5992            segvn_vpage(seg);
5993            svd->pageprot = 1;
5994            evp = &svd->vpage[seg_page(seg, addr + len)];
5995            for (svp = &svd->vpage[seg_page(seg, addr)]; svp < evp; svp++) {
5996                    VPP_SETPROT(svp, prot);
5997            }
5998    }
6000    if (unload_done) {
```



```
6001                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6002                    return (0);
6003            }
6005            if (((prot & PROT_WRITE) != 0 &&
6006                (svd->vp != NULL || svd->type == MAP_PRIVATE)) ||
6007                (prot & ~PROT_USER) == PROT_NONE) {
6008                    /*
6009                     * Either private or shared data with write access (in
6010                     * which case we need to throw out all former translations
6011                     * so that we get the right translations set up on fault
6012                     * and we don't allow write access to any copy-on-write pages
6013                     * that might be around or to prevent write access to pages
6014                     * representing holes in a file), or we don't have permission
6015                     * to access the memory at all (in which case we have to
6016                     * unload any current translations that might exist).
6017                     */
6018                    hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
6019            } else {
6020                    /*
6021                     * A shared mapping or a private mapping in which write
6022                     * protection is going to be denied - just change all the
6023                     * protections over the range of addresses in question.
6024                     * segvn does not support any other attributes other
6025                     * than prot so we can use hat_chgattr.
6026                     */
6027                    hat_chgattr(seg->s_as->a_hat, addr, len, prot);
6028            }
6030            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6032            return (0);
6033    }
6035    /*
6036     * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
6037     * to determine if the seg is capable of mapping the requested szc.
6038     */
6039    static int
6040    segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
6041    {
6042            struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6043            struct segvn_data *nsvd;
6044            struct anon_map *amp = svd->amp;
6045            struct seg *nseg;
6046            caddr_t eaddr = addr + len, a;
6047            size_t pgsz = page_get_pagesize(szc);
6048            pgcnt_t pgcnt = page_get_pagecnt(szc);
6049            int err;
6050            u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);
6052            ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6053            ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);
6055            if (seg->s_szc == szc || segvn_lpg_disable != 0) {
6056                    return (0);
6057            }
6059            /*
6060             * addr should always be pgsz aligned but eaddr may be misaligned if
6061             * it's at the end of the segment.
6062             *
6063             * XXX we should assert this condition since as_setpagesize() logic
6064             * guarantees it.
6065             */
6066            if (!IS_P2ALIGNED(addr, pgsz) ||
```

```
6067             (!IS_P2ALIGNED(eaddr, pgsz) &&
6068             eaddr != seg->s_base + seg->s_size)) {

6070                 segvn_setpgsz_align_err++;
6071                 return (EINVAL);
6072         }

6074         if (amp != NULL && svd->type == MAP_SHARED) {
6075                 ulong_t an_idx = svd->anon_index + seg_page(seg, addr);
6076                 if (!IS_P2ALIGNED(an_idx, pgcnt)) {

6078                         segvn_setpgsz_anon_align_err++;
6079                         return (EINVAL);
6080                 }
6081         }

6083         if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
6084             szc > segvn_maxpgszc) {
6085                 return (EINVAL);
6086         }

6088         /* paranoid check */
6089         if (svd->vp != NULL &&
6090             (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
6091                 return (EINVAL);
6092         }

6094         if (seg->s_szc == 0 && svd->vp != NULL &&
6095             map_addr_vacalign_check(addr, off)) {
6096                 return (EINVAL);
6097         }

6099         /*
6100          * Check that protections are the same within new page
6101          * size boundaries.
6102          */
6103         if (svd->pageprot) {
6104                 for (a = addr; a < eaddr; a += pgsz) {
6105                         if ((a + pgsz) > eaddr) {
6106                                 if (!sameprot(seg, a, eaddr - a)) {
6107                                         return (EINVAL);
6108                                 }
6109                         } else {
6110                                 if (!sameprot(seg, a, pgsz)) {
6111                                         return (EINVAL);
6112                                 }
6113                         }
6114                 }
6115         }

6117         /*
6118          * Since we are changing page size we first have to flush
6119          * the cache. This makes sure all the pagelock calls have
6120          * to recheck protections.
6121          */
6122         if (svd->softlockcnt > 0) {
6123                 ASSERT(svd->tr_state == SEGVN_TR_OFF);

6125                 /*
6126                  * If this is shared segment non 0 softlockcnt
6127                  * means locked pages are still in use.
6128                  */
6129                 if (svd->type == MAP_SHARED) {
6130                         return (EAGAIN);
6131                 }
```



```
6133                 /*
6134                  * Since we do have the segvn writers lock nobody can fill
6135                  * the cache with entries belonging to this seg during
6136                  * the purge. The flush either succeeds or we still have
6137                  * pending I/Os.
6138                  */
6139                 segvn_purge(seg);
6140                 if (svd->softlockcnt > 0) {
6141                         return (EAGAIN);
6142                 }
6143         }

6145         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6146                 ASSERT(svd->amp == NULL);
6147                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
6148                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6149                     HAT_REGION_TEXT);
6150                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
6151         } else if (svd->tr_state == SEGVN_TR_INIT) {
6152                 svd->tr_state = SEGVN_TR_OFF;
6153         } else if (svd->tr_state == SEGVN_TR_ON) {
6154                 ASSERT(svd->amp != NULL);
6155                 segvn_textunrepl(seg, 1);
6156                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6157                 amp = NULL;
6158         }

6160         /*
6161          * Operation for sub range of existing segment.
6162          */
6163         if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6164                 if (szc < seg->s_szc) {
6165                         VM_STAT_ADD(segvnvmstats.demoterange[2]);
6166                         err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6167                         if (err == 0) {
6168                                 return (IE_RETRY);
6169                         }
6170                         if (err == ENOMEM) {
6171                                 return (IE_NOMEM);
6172                         }
6173                         return (err);
6174                 }
6175                 if (addr != seg->s_base) {
6176                         nseg = segvn_split_seg(seg, addr);
6177                         if (eaddr != (nseg->s_base + nseg->s_size)) {
6178                                 /* eaddr is szc aligned */
6179                                 (void) segvn_split_seg(nseg, eaddr);
6180                         }
6181                         return (IE_RETRY);
6182                 }
6183                 if (eaddr != (seg->s_base + seg->s_size)) {
6184                         /* eaddr is szc aligned */
6185                         (void) segvn_split_seg(seg, eaddr);
6186                 }
6187                 return (IE_RETRY);
6188         }

6190         /*
6191          * Break any low level sharing and reset seg->s_szc to 0.
6192          */
6193         if ((err = segvn_clrszc(seg)) != 0) {
6194                 if (err == ENOMEM) {
6195                         err = IE_NOMEM;
6196                 }
6197                 return (err);
6198         }
```

```
6199         ASSERT(seg->s_szc == 0);
6200
6201         /*
6202          * If the end of the current segment is not pgsz aligned
6203          * then attempt to concatenate with the next segment.
6204          */
6205         if (!IS_P2ALIGNED(eaddr, pgsz)) {
6206                 nseg = AS_SEGNEXT(seg->s_as, seg);
6207                 if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6208                         return (ENOMEM);
6209                 }
6210                 if (nseg->s_ops != &segvn_ops) {
6211                         return (EINVAL);
6212                 }
6213                 nsvd = (struct segvn_data *)nseg->s_data;
6214                 if (nsvd->softlockcnt > 0) {
6215                         /*
6216                          * If this is shared segment non 0 softlockcnt
6217                          * means locked pages are still in use.
6218                          */
6219                         if (nsvd->type == MAP_SHARED) {
6220                                 return (EAGAIN);
6221                         }
6222                         segvn_purge(nseg);
6223                         if (nsvd->softlockcnt > 0) {
6224                                 return (EAGAIN);
6225                         }
6226                 }
6227                 err = segvn_clrszc(nseg);
6228                 if (err == ENOMEM) {
6229                         err = IE_NOMEM;
6230                 }
6231                 if (err != 0) {
6232                         return (err);
6233                 }
6234                 ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6235                 err = segvn_concat(seg, nseg, 1);
6236                 if (err == -1) {
6237                         return (EINVAL);
6238                 }
6239                 if (err == -2) {
6240                         return (IE_NOMEM);
6241                 }
6242                 return (IE_RETRY);
6243         }
6244
6245         /*
6246          * May need to re-align anon array to
6247          * new szc.
6248          */
6249         if (amp != NULL) {
6250                 if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6251                         struct anon_hdr *nahp;
6252
6253                         ASSERT(svd->type == MAP_PRIVATE);
6254
6255                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6256                         ASSERT(amp->refcnt == 1);
6257                         nahp = anon_create(btop(amp->size), ANON_NOSLEEP);
6258                         if (nahp == NULL) {
6259                                 ANON_LOCK_EXIT(&amp->a_rwlock);
6260                                 return (IE_NOMEM);
6261                         }
6262                         if (anon_copy_ptr(amp->ahp, svd->anon_index,
6263                             nahp, 0, btop(seg->s_size), ANON_NOSLEEP)) {
6264                                 anon_release(nahp, btop(amp->size));
```



```
6265                                 ANON_LOCK_EXIT(&amp->a_rwlock);
6266                                 return (IE_NOMEM);
6267                         }
6268                         anon_release(amp->ahp, btop(amp->size));
6269                         amp->ahp = nahp;
6270                         svd->anon_index = 0;
6271                         ANON_LOCK_EXIT(&amp->a_rwlock);
6272                 }
6273         }
6274         if (svd->vp != NULL && szc != 0) {
6275                 struct vattr va;
6276                 u_offset_t eoffpage = svd->offset;
6277                 va.va_mask = AT_SIZE;
6278                 eoffpage += seg->s_size;
6279                 eoffpage = btopr(eoffpage);
6280                 if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
6281                         segvn_setpgsz_getattr_err++;
6282                         return (EINVAL);
6283                 }
6284                 if (btopr(va.va_size) < eoffpage) {
6285                         segvn_setpgsz_eof_err++;
6286                         return (EINVAL);
6287                 }
6288                 if (amp != NULL) {
6289                         /*
6290                          * anon_fill_cow_holes() may call VOP_GETPAGE().
6291                          * don't take anon map lock here to avoid holding it
6292                          * across VOP_GETPAGE() calls that may call back into
6293                          * segvn for klsutering checks. We don't really need
6294                          * anon map lock here since it's a private segment and
6295                          * we hold as level lock as writers.
6296                          */
6297                         if ((err = anon_fill_cow_holes(seg, seg->s_base,
6298                             amp->ahp, svd->anon_index, svd->vp, svd->offset,
6299                             seg->s_size, szc, svd->prot, svd->vpage,
6300                             svd->cred)) != 0) {
6301                                 return (EINVAL);
6302                         }
6303                 }
6304                 segvn_setvnode_mpss(svd->vp);
6305         }
6306
6307         if (amp != NULL) {
6308                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6309                 if (svd->type == MAP_PRIVATE) {
6310                         amp->a_szc = szc;
6311                 } else if (szc > amp->a_szc) {
6312                         amp->a_szc = szc;
6313                 }
6314                 ANON_LOCK_EXIT(&amp->a_rwlock);
6315         }
6316
6317         seg->s_szc = szc;
6318
6319         return (0);
6320 }
6321
6322 static int
6323 segvn_clrszc(struct seg *seg)
6324 {
6325         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6326         struct anon_map *amp = svd->amp;
6327         size_t pgsz;
6328         pgcnt_t pages;
6329         int err = 0;
6330         caddr_t a = seg->s_base;
```

```
6331         caddr_t ea = a + seg->s_size;
6332         ulong_t an_idx = svd->anon_index;
6333         vnode_t *vp = svd->vp;
6334         struct vpage *vpage = svd->vpage;
6335         page_t *anon_pl[1 + 1], *pp;
6336         struct anon *ap, *oldap;
6337         uint_t prot = svd->prot, vpprot;
6338         int pageflag = 0;

6340         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
6341             SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
6342         ASSERT(svd->softlockcnt == 0);

6344         if (vp == NULL && amp == NULL) {
6345                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6346                 seg->s_szc = 0;
6347                 return (0);
6348         }

6350         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6351                 ASSERT(svd->amp == NULL);
6352                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
6353                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6354                     HAT_REGION_TEXT);
6355                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
6356         } else if (svd->tr_state == SEGVN_TR_ON) {
6357                 ASSERT(svd->amp != NULL);
6358                 segvn_textunrepl(seg, 1);
6359                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6360                 amp = NULL;
6361         } else {
6362                 if (svd->tr_state != SEGVN_TR_OFF) {
6363                         ASSERT(svd->tr_state == SEGVN_TR_INIT);
6364                         svd->tr_state = SEGVN_TR_OFF;
6365                 }

6367                 /*
6368                  * do HAT_UNLOAD_UNMAP since we are changing the pagesize.
6369                  * unload argument is 0 when we are freeing the segment
6370                  * and unload was already done.
6371                  */
6372                 hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
6373                     HAT_UNLOAD_UNMAP);
6374         }

6376         if (amp == NULL || svd->type == MAP_SHARED) {
6377                 seg->s_szc = 0;
6378                 return (0);
6379         }

6381         pgsz = page_get_pagesize(seg->s_szc);
6382         pages = btop(pgsz);

6384         /*
6385          * XXX anon rwlock is not really needed because this is a
6386          * private segment and we are writers.
6387          */
6388         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);

6390         for (; a < ea; a += pgsz, an_idx += pages) {
6391                 if ((oldap = anon_get_ptr(amp->ahp, an_idx)) != NULL) {
6392                         ASSERT(vpage != NULL || svd->pageprot == 0);
6393                         if (vpage != NULL) {
6394                                 ASSERT(sameprot(seg, a, pgsz));
6395                                 prot = VPP_PROT(vpage);
6396                                 pageflag = VPP_ISPPLOCK(vpage) ? LOCK_PAGE : 0;
```



```
6397                         }
6398                         if (seg->s_szc != 0) {
6399                                 ASSERT(vp == NULL || anon_pages(amp->ahp,
6400                                     an_idx, pages) == pages);
6401                                 if ((err = anon_map_demotepages(amp, an_idx,
6402                                     seg, a, prot, vpage, svd->cred)) != 0) {
6403                                         goto out;
6404                                 }
6405                         } else {
6406                                 if (oldap->an_refcnt == 1) {
6407                                         continue;
6408                                 }
6409                                 if ((err = anon_getpage(&oldap, &vpprot,
6410                                     anon_pl, PAGESIZE, seg, a, S_READ,
6411                                     svd->cred))) {
6412                                         goto out;
6413                                 }
6414                                 if ((pp = anon_private(&ap, seg, a, prot,
6415                                     anon_pl[0], pageflag, svd->cred)) == NULL) {
6416                                         err = ENOMEM;
6417                                         goto out;
6418                                 }
6419                                 anon_decref(oldap);
6420                                 (void) anon_set_ptr(amp->ahp, an_idx, ap,
6421                                     ANON_SLEEP);
6422                                 page_unlock(pp);
6423                         }
6424                 }
6425                 vpage = (vpage == NULL) ? NULL : vpage + pages;
6426         }

6428         amp->a_szc = 0;
6429         seg->s_szc = 0;
6430 out:
6431         ANON_LOCK_EXIT(&amp->a_rwlock);
6432         return (err);
6433 }

6435 static int
6436 segvn_claim_pages(
6437         struct seg *seg,
6438         struct vpage *svp,
6439         u_offset_t off,
6440         ulong_t anon_idx,
6441         uint_t prot)
6442 {
6443         pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
6444         size_t ppasize = (pgcnt + 1) * sizeof (page_t *);
6445         page_t  **ppa;
6446         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6447         struct anon_map *amp = svd->amp;
6448         struct vpage *evp = svp + pgcnt;
6449         caddr_t addr = ((uintptr_t)(svp - svd->vpage) << PAGESHIFT)
6450             + seg->s_base;
6451         struct anon *ap;
6452         struct vnode *vp = svd->vp;
6453         page_t *pp;
6454         pgcnt_t pg_idx, i;
6455         int err = 0;
6456         anoff_t aoff;
6457         int anon = (amp != NULL) ? 1 : 0;

6459         ASSERT(svd->type == MAP_PRIVATE);
6460         ASSERT(svd->vpage != NULL);
6461         ASSERT(seg->s_szc != 0);
6462         ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
```

```
6463         ASSERT(amp == NULL || IS_P2ALIGNED(anon_idx, pgcnt));
6464         ASSERT(sameprot(seg, addr, pgcnt << PAGESHIFT));
6465
6466         if (VPP_PROT(svp) == prot)
6467                 return (1);
6468         if (!((VPP_PROT(svp) ^ prot) & PROT_WRITE))
6469                 return (1);
6470
6471         ppa = kmem_alloc(ppasize, KM_SLEEP);
6472         if (anon && vp != NULL) {
6473                 if (anon_get_ptr(amp->ahp, anon_idx) == NULL) {
6474                         anon = 0;
6475                         ASSERT(!anon_pages(amp->ahp, anon_idx, pgcnt));
6476                 }
6477                 ASSERT(!anon ||
6478                     anon_pages(amp->ahp, anon_idx, pgcnt) == pgcnt);
6479         }
6480
6481         for (*ppa = NULL, pg_idx = 0; svp < evp; svp++, anon_idx++) {
6482                 if (!VPP_ISPPLOCK(svp))
6483                         continue;
6484                 if (anon) {
6485                         ap = anon_get_ptr(amp->ahp, anon_idx);
6486                         if (ap == NULL) {
6487                                 panic("segvn_claim_pages: no anon slot");
6488                         }
6489                         swap_xlate(ap, &vp, &aoff);
6490                         off = (u_offset_t)aoff;
6491                 }
6492                 ASSERT(vp != NULL);
6493                 if ((pp = page_lookup(vp,
6494                     (u_offset_t)off, SE_SHARED)) == NULL) {
6495                         panic("segvn_claim_pages: no page");
6496                 }
6497                 ppa[pg_idx++] = pp;
6498                 off += PAGESIZE;
6499         }
6500
6501         if (ppa[0] == NULL) {
6502                 kmem_free(ppa, ppasize);
6503                 return (1);
6504         }
6505
6506         ASSERT(pg_idx <= pgcnt);
6507         ppa[pg_idx] = NULL;
6508
6509
6510         /* Find each large page within ppa, and adjust its claim */
6511
6512         /* Does ppa cover a single large page? */
6513         if (ppa[0]->p_szc == seg->s_szc) {
6514                 if (prot & PROT_WRITE)
6515                         err = page_addclaim_pages(ppa);
6516                 else
6517                         err = page_subclaim_pages(ppa);
6518         } else {
6519                 for (i = 0; ppa[i]; i += pgcnt) {
6520                         ASSERT(IS_P2ALIGNED(page_pptonum(ppa[i]), pgcnt));
6521                         if (prot & PROT_WRITE)
6522                                 err = page_addclaim_pages(&ppa[i]);
6523                         else
6524                                 err = page_subclaim_pages(&ppa[i]);
6525                         if (err == 0)
6526                                 break;
6527                 }
6528         }
```



```
6529
6530         for (i = 0; i < pg_idx; i++) {
6531                 ASSERT(ppa[i] != NULL);
6532                 page_unlock(ppa[i]);
6533         }
6534
6535         kmem_free(ppa, ppasize);
6536         return (err);
6537 }
6538
6539 /*
6540  * Returns right (upper address) segment if split occurred.
6541  * If the address is equal to the beginning or end of its segment it returns
6542  * the current segment.
6543  */
6544 static struct seg *
6545 segvn_split_seg(struct seg *seg, caddr_t addr)
6546 {
6547         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6548         struct seg *nseg;
6549         size_t nsize;
6550         struct segvn_data *nsvd;
6551
6552         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6553         ASSERT(svd->tr_state == SEGVN_TR_OFF);
6554
6555         ASSERT(addr >= seg->s_base);
6556         ASSERT(addr <= seg->s_base + seg->s_size);
6557         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6558
6559         if (addr == seg->s_base || addr == seg->s_base + seg->s_size)
6560                 return (seg);
6561
6562         nsize = seg->s_base + seg->s_size - addr;
6563         seg->s_size = addr - seg->s_base;
6564         nseg = seg_alloc(seg->s_as, addr, nsize);
6565         ASSERT(nseg != NULL);
6566         nseg->s_ops = seg->s_ops;
6567         nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
6568         nseg->s_data = (void *)nsvd;
6569         nseg->s_szc = seg->s_szc;
6570         *nsvd = *svd;
6571         ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6572         nsvd->seg = nseg;
6573         rw_init(&nsvd->lock, NULL, RW_DEFAULT, NULL);
6574
6575         if (nsvd->vp != NULL) {
6576                 VN_HOLD(nsvd->vp);
6577                 nsvd->offset = svd->offset +
6578                     (uintptr_t)(nseg->s_base - seg->s_base);
6579                 if (nsvd->type == MAP_SHARED)
6580                         lgrp_shm_policy_init(NULL, nsvd->vp);
6581         } else {
6582                 /*
6583                  * The offset for an anonymous segment has no signifigance in
6584                  * terms of an offset into a file. If we were to use the above
6585                  * calculation instead, the structures read out of
6586                  * /proc/<pid>/xmap would be more difficult to decipher since
6587                  * it would be unclear whether two seemingly contiguous
6588                  * prxmap_t structures represented different segments or a
6589                  * single segment that had been split up into multiple prxmap_t
6590                  * structures (e.g. if some part of the segment had not yet
6591                  * been faulted in).
6592                  */
6593                 nsvd->offset = 0;
6594         }
```

```
6596         ASSERT(svd->softlockcnt == 0);
6597         ASSERT(svd->softlockcnt_sbase == 0);
6598         ASSERT(svd->softlockcnt_send == 0);
6599         crhold(svd->cred);

6601         if (svd->vpage != NULL) {
6602                 size_t bytes = vpgtob(seg_pages(seg));
6603                 size_t nbytes = vpgtob(seg_pages(nseg));
6604                 struct vpage *ovpage = svd->vpage;

6606                 svd->vpage = kmem_alloc(bytes, KM_SLEEP);
6607                 bcopy(ovpage, svd->vpage, bytes);
6608                 nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);
6609                 bcopy(ovpage + seg_pages(seg), nsvd->vpage, nbytes);
6610                 kmem_free(ovpage, bytes + nbytes);
6611         }
6612         if (svd->amp != NULL && svd->type == MAP_PRIVATE) {
6613                 struct anon_map *oamp = svd->amp, *namp;
6614                 struct anon_hdr *nahp;

6616                 ANON_LOCK_ENTER(&oamp->a_rwlock, RW_WRITER);
6617                 ASSERT(oamp->refcnt == 1);
6618                 nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
6619                 (void) anon_copy_ptr(oamp->ahp, svd->anon_index,
6620                     nahp, 0, btop(seg->s_size), ANON_SLEEP);

6622                 namp = anonmap_alloc(nseg->s_size, 0, ANON_SLEEP);
6623                 namp->a_szc = nseg->s_szc;
6624                 (void) anon_copy_ptr(oamp->ahp,
6625                     svd->anon_index + btop(seg->s_size),
6626                     namp->ahp, 0, btop(nseg->s_size), ANON_SLEEP);
6627                 anon_release(oamp->ahp, btop(oamp->size));
6628                 oamp->ahp = nahp;
6629                 oamp->size = seg->s_size;
6630                 svd->anon_index = 0;
6631                 nsvd->amp = namp;
6632                 nsvd->anon_index = 0;
6633                 ANON_LOCK_EXIT(&oamp->a_rwlock);
6634         } else if (svd->amp != NULL) {
6635                 pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
6636                 ASSERT(svd->amp == nsvd->amp);
6637                 ASSERT(seg->s_szc <= svd->amp->a_szc);
6638                 nsvd->anon_index = svd->anon_index + seg_pages(seg);
6639                 ASSERT(IS_P2ALIGNED(nsvd->anon_index, pgcnt));
6640                 ANON_LOCK_ENTER(&svd->amp->a_rwlock, RW_WRITER);
6641                 svd->amp->refcnt++;
6642                 ANON_LOCK_EXIT(&svd->amp->a_rwlock);
6643         }

6645         /*
6646          * Split the amount of swap reserved.
6647          */
6648         if (svd->swresv) {
6649                 /*
6650                  * For MAP_NORESERVE, only allocate swap reserve for pages
6651                  * being used.  Other segments get enough to cover whole
6652                  * segment.
6653                  */
6654                 if (svd->flags & MAP_NORESERVE) {
6655                         size_t  oswresv;

6657                         ASSERT(svd->amp);
6658                         oswresv = svd->swresv;
6659                         svd->swresv = ptob(anon_pages(svd->amp->ahp,
6660                             svd->anon_index, btop(seg->s_size)));
```



```
6661                         nsvd->swresv = ptob(anon_pages(nsvd->amp->ahp,
6662                             nsvd->anon_index, btop(nseg->s_size)));
6663                         ASSERT(oswresv >= (svd->swresv + nsvd->swresv));
6664                 } else {
6665                         if (svd->pageswap) {
6666                                 svd->swresv = segvn_count_swap_by_vpages(seg);
6667                                 ASSERT(nsvd->swresv >= svd->swresv);
6668                                 nsvd->swresv -= svd->swresv;
6669                         } else {
6670                                 ASSERT(svd->swresv == seg->s_size +
6671                                     nseg->s_size);
6672                                 svd->swresv = seg->s_size;
6673                                 nsvd->swresv = nseg->s_size;
6674                         }
6675                 }
6676         }

6678         return (nseg);
6679 }

6681 /*
6682  * called on memory operations (unmap, setprot, setpagesize) for a subset
6683  * of a large page segment to either demote the memory range (SDR_RANGE)
6684  * or the ends (SDR_END) by addr/len.
6685  *
6686  * returns 0 on success. returns errno, including ENOMEM, on failure.
6687  */
6688 static int
6689 segvn_demote_range(
6690         struct seg *seg,
6691         caddr_t addr,
6692         size_t len,
6693         int flag,
6694         uint_t szcvec)
6695 {
6696         caddr_t eaddr = addr + len;
6697         caddr_t lpgaddr, lpgeaddr;
6698         struct seg *nseg;
6699         struct seg *badseg1 = NULL;
6700         struct seg *badseg2 = NULL;
6701         size_t pgsz;
6702         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6703         int err;
6704         uint_t szc = seg->s_szc;
6705         uint_t tszcvec;

6707         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6708         ASSERT(svd->tr_state == SEGVN_TR_OFF);
6709         ASSERT(szc != 0);
6710         pgsz = page_get_pagesize(szc);
6711         ASSERT(seg->s_base != addr || seg->s_size != len);
6712         ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);
6713         ASSERT(svd->softlockcnt == 0);
6714         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6715         ASSERT(szcvec == 0 || (flag == SDR_END && svd->type == MAP_SHARED));

6717         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
6718         ASSERT(flag == SDR_RANGE || eaddr < lpgeaddr || addr > lpgaddr);
6719         if (flag == SDR_RANGE) {
6720                 /* demote entire range */
6721                 badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
6722                 (void) segvn_split_seg(nseg, lpgeaddr);
6723                 ASSERT(badseg1->s_base == lpgaddr);
6724                 ASSERT(badseg1->s_size == lpgeaddr - lpgaddr);
6725         } else if (addr != lpgaddr) {
6726                 ASSERT(flag == SDR_END);
```

```
6727                badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
6728                if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz &&
6729                    eaddr < lpgaddr + 2 * pgsz) {
6730                        (void) segvn_split_seg(nseg, lpgeaddr);
6731                        ASSERT(badseg1->s_base == lpgaddr);
6732                        ASSERT(badseg1->s_size == 2 * pgsz);
6733                } else {
6734                        nseg = segvn_split_seg(nseg, lpgaddr + pgsz);
6735                        ASSERT(badseg1->s_base == lpgaddr);
6736                        ASSERT(badseg1->s_size == pgsz);
6737                        if (eaddr != lpgeaddr && eaddr > lpgaddr + pgsz) {
6738                                ASSERT(lpgeaddr - lpgaddr > 2 * pgsz);
6739                                nseg = segvn_split_seg(nseg, lpgeaddr - pgsz);
6740                                badseg2 = nseg;
6741                                (void) segvn_split_seg(nseg, lpgeaddr);
6742                                ASSERT(badseg2->s_base == lpgeaddr - pgsz);
6743                                ASSERT(badseg2->s_size == pgsz);
6744                        }
6745                }
6746        } else {
6747                ASSERT(flag == SDR_END);
6748                ASSERT(eaddr < lpgeaddr);
6749                badseg1 = nseg = segvn_split_seg(seg, lpgeaddr - pgsz);
6750                (void) segvn_split_seg(nseg, lpgeaddr);
6751                ASSERT(badseg1->s_base == lpgeaddr - pgsz);
6752                ASSERT(badseg1->s_size == pgsz);
6753        }

6755        ASSERT(badseg1 != NULL);
6756        ASSERT(badseg1->s_szc == szc);
6757        ASSERT(flag == SDR_RANGE || badseg1->s_size == pgsz ||
6758            badseg1->s_size == 2 * pgsz);
6759        ASSERT(sameprot(badseg1, badseg1->s_base, pgsz));
6760        ASSERT(badseg1->s_size == pgsz ||
6761            sameprot(badseg1, badseg1->s_base + pgsz, pgsz));
6762        if (err = segvn_clrszc(badseg1)) {
6763                return (err);
6764        }
6765        ASSERT(badseg1->s_szc == 0);

6767        if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
6768                uint_t tszc = highbit(tszcvec) - 1;
6769                caddr_t ta = MAX(addr, badseg1->s_base);
6770                caddr_t te;
6771                size_t tpgsz = page_get_pagesize(tszc);

6773                ASSERT(svd->type == MAP_SHARED);
6774                ASSERT(flag == SDR_END);
6775                ASSERT(tszc < szc && tszc > 0);

6777                if (eaddr > badseg1->s_base + badseg1->s_size) {
6778                        te = badseg1->s_base + badseg1->s_size;
6779                } else {
6780                        te = eaddr;
6781                }

6783                ASSERT(ta <= te);
6784                badseg1->s_szc = tszc;
6785                if (!IS_P2ALIGNED(ta, tpgsz) || !IS_P2ALIGNED(te, tpgsz)) {
6786                        if (badseg2 != NULL) {
6787                                err = segvn_demote_range(badseg1, ta, te - ta,
6788                                    SDR_END, tszcvec);
6789                                if (err != 0) {
6790                                        return (err);
6791                                }
6792                        } else {
```



```
6793                                return (segvn_demote_range(badseg1, ta,
6794                                    te - ta, SDR_END, tszcvec));
6795                        }
6796                }
6797        }

6799        if (badseg2 == NULL)
6800                return (0);
6801        ASSERT(badseg2->s_szc == szc);
6802        ASSERT(badseg2->s_size == pgsz);
6803        ASSERT(sameprot(badseg2, badseg2->s_base, badseg2->s_size));
6804        if (err = segvn_clrszc(badseg2)) {
6805                return (err);
6806        }
6807        ASSERT(badseg2->s_szc == 0);

6809        if (szc > 1 && (tszcvec = P2PHASE(szcvec, 1 << szc)) > 1) {
6810                uint_t tszc = highbit(tszcvec) - 1;
6811                size_t tpgsz = page_get_pagesize(tszc);

6813                ASSERT(svd->type == MAP_SHARED);
6814                ASSERT(flag == SDR_END);
6815                ASSERT(tszc < szc && tszc > 0);
6816                ASSERT(badseg2->s_base > addr);
6817                ASSERT(eaddr > badseg2->s_base);
6818                ASSERT(eaddr < badseg2->s_base + badseg2->s_size);

6820                badseg2->s_szc = tszc;
6821                if (!IS_P2ALIGNED(eaddr, tpgsz)) {
6822                        return (segvn_demote_range(badseg2, badseg2->s_base,
6823                            eaddr - badseg2->s_base, SDR_END, tszcvec));
6824                }
6825        }

6827        return (0);
6828 }

6830 static int
6831 segvn_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
6832 {
6833        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6834        struct vpage *vp, *evp;

6836        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6838        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
6839        /*
6840         * If segment protection can be used, simply check against them.
6841         */
6842        if (svd->pageprot == 0) {
6843                int err;

6845                err = ((svd->prot & prot) != prot) ? EACCES : 0;
6846                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6847                return (err);
6848        }

6850        /*
6851         * Have to check down to the vpage level.
6852         */
6853        evp = &svd->vpage[seg_page(seg, addr + len)];
6854        for (vp = &svd->vpage[seg_page(seg, addr)]; vp < evp; vp++) {
6855                if ((VPP_PROT(vp) & prot) != prot) {
6856                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6857                        return (EACCES);
6858                }
```

```
6859         }
6860         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6861         return (0);
6862 }

6864 static int
6865 segvn_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *protv)
6866 {
6867         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6868         size_t pgno = seg_page(seg, addr + len) - seg_page(seg, addr) + 1;

6870         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6872         if (pgno != 0) {
6873                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
6874                 if (svd->pageprot == 0) {
6875                         do {
6876                                 protv[--pgno] = svd->prot;
6877                         } while (pgno != 0);
6878                 } else {
6879                         size_t pgoff = seg_page(seg, addr);

6881                         do {
6882                                 pgno--;
6883                                 protv[pgno] = VPP_PROT(&svd->vpage[pgno+pgoff]);
6884                         } while (pgno != 0);
6885                 }
6886                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6887         }
6888         return (0);
6889 }

6891 static u_offset_t
6892 segvn_getoffset(struct seg *seg, caddr_t addr)
6893 {
6894         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6896         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6898         return (svd->offset + (uintptr_t)(addr - seg->s_base));
6899 }

6901 /*ARGSUSED*/
6902 static int
6903 segvn_gettype(struct seg *seg, caddr_t addr)
6904 {
6905         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6907         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6909         return (svd->type | (svd->flags & (MAP_NORESERVE | MAP_TEXT |
6910             MAP_INITDATA)));
6911 }

6913 /*ARGSUSED*/
6914 static int
6915 segvn_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
6916 {
6917         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

6919         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

6921         *vpp = svd->vp;
6922         return (0);
6923 }
```



```
6925 /*
6926  * Check to see if it makes sense to do kluster/read ahead to
6927  * addr + delta relative to the mapping at addr.  We assume here
6928  * that delta is a signed PAGESIZE'd multiple (which can be negative).
6929  *
6930  * For segvn, we currently "approve" of the action if we are
6931  * still in the segment and it maps from the same vp/off,
6932  * or if the advice stored in segvn_data or vpages allows it.
6933  * Currently, klustering is not allowed only if MADV_RANDOM is set.
6934  */
6935 static int
6936 segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
6937 {
6938         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6939         struct anon *oap, *ap;
6940         ssize_t pd;
6941         size_t page;
6942         struct vnode *vp1, *vp2;
6943         u_offset_t off1, off2;
6944         struct anon_map *amp;

6946         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6947         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
6948             SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

6950         if (addr + delta < seg->s_base ||
6951             addr + delta >= (seg->s_base + seg->s_size))
6952                 return (-1);            /* exceeded segment bounds */

6954         pd = delta / (ssize_t)PAGESIZE; /* divide to preserve sign bit */
6955         page = seg_page(seg, addr);

6957         /*
6958          * Check to see if either of the pages addr or addr + delta
6959          * have advice set that prevents klustering (if MADV_RANDOM advice
6960          * is set for entire segment, or MADV_SEQUENTIAL is set and delta
6961          * is negative).
6962          */
6963         if (svd->advice == MADV_RANDOM ||
6964             svd->advice == MADV_SEQUENTIAL && delta < 0)
6965                 return (-1);
6966         else if (svd->pageadvice && svd->vpage) {
6967                 struct vpage *bvpp, *evpp;

6969                 bvpp = &svd->vpage[page];
6970                 evpp = &svd->vpage[page + pd];
6971                 if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
6972                     VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
6973                         return (-1);
6974                 if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
6975                     VPP_ADVICE(evpp) == MADV_RANDOM)
6976                         return (-1);
6977         }

6979         if (svd->type == MAP_SHARED)
6980                 return (0);             /* shared mapping - all ok */

6982         if ((amp = svd->amp) == NULL)
6983                 return (0);             /* off original vnode */

6985         page += svd->anon_index;

6987         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

6989         oap = anon_get_ptr(amp->ahp, page);
6990         ap = anon_get_ptr(amp->ahp, page + pd);
```

```
6992         ANON_LOCK_EXIT(&amp->a_rwlock);

6994         if ((oap == NULL && ap != NULL) || (oap != NULL && ap == NULL)) {
6995                 return (-1);            /* one with and one without an anon */
6996         }

6998         if (oap == NULL) {              /* implies that ap == NULL */
6999                 return (0);             /* off original vnode */
7000         }

7002         /*
7003          * Now we know we have two anon pointers - check to
7004          * see if they happen to be properly allocated.
7005          */

7007         /*
7008          * XXX We cheat here and don't lock the anon slots. We can't because
7009          * we may have been called from the anon layer which might already
7010          * have locked them. We are holding a refcnt on the slots so they
7011          * can't disappear. The worst that will happen is we'll get the wrong
7012          * names (vp, off) for the slots and make a poor klustering decision.
7013          */
7014         swap_xlate(ap, &vp1, &off1);
7015         swap_xlate(oap, &vp2, &off2);


7018         if (!VOP_CMP(vp1, vp2, NULL) || off1 - off2 != delta)
7019                 return (-1);
7020         return (0);
7021 }

7023 /*
7024  * Swap the pages of seg out to secondary storage, returning the
7025  * number of bytes of storage freed.
7026  *
7027  * The basic idea is first to unload all translations and then to call
7028  * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
7029  * swap device.  Pages to which other segments have mappings will remain
7030  * mapped and won't be swapped.  Our caller (as_swapout) has already
7031  * performed the unloading step.
7032  *
7033  * The value returned is intended to correlate well with the process's
7034  * memory requirements.  However, there are some caveats:
7035  * 1)   When given a shared segment as argument, this routine will
7036  *      only succeed in swapping out pages for the last sharer of the
7037  *      segment.  (Previous callers will only have decremented mapping
7038  *      reference counts.)
7039  * 2)   We assume that the hat layer maintains a large enough translation
7040  *      cache to capture process reference patterns.
7041  */
7042 static size_t
7043 segvn_swapout(struct seg *seg)
7044 {
7045         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7046         struct anon_map *amp;
7047         pgcnt_t pgcnt = 0;
7048         pgcnt_t npages;
7049         pgcnt_t page;
7050         ulong_t anon_index;

7052         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7054         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
7055         /*
7056          * Find pages unmapped by our caller and force them
```



```
7057          * out to the virtual swap device.
7058          */
7059         if ((amp = svd->amp) != NULL)
7060                 anon_index = svd->anon_index;
7061         npages = seg->s_size >> PAGESHIFT;
7062         for (page = 0; page < npages; page++) {
7063                 page_t *pp;
7064                 struct anon *ap;
7065                 struct vnode *vp;
7066                 u_offset_t off;
7067                 anon_sync_obj_t cookie;

7069                 /*
7070                  * Obtain <vp, off> pair for the page, then look it up.
7071                  *
7072                  * Note that this code is willing to consider regular
7073                  * pages as well as anon pages.  Is this appropriate here?
7074                  */
7075                 ap = NULL;
7076                 if (amp != NULL) {
7077                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7078                         if (anon_array_try_enter(amp, anon_index + page,
7079                             &cookie)) {
7080                                 ANON_LOCK_EXIT(&amp->a_rwlock);
7081                                 continue;
7082                         }
7083                         ap = anon_get_ptr(amp->ahp, anon_index + page);
7084                         if (ap != NULL) {
7085                                 swap_xlate(ap, &vp, &off);
7086                         } else {
7087                                 vp = svd->vp;
7088                                 off = svd->offset + ptob(page);
7089                         }
7090                         anon_array_exit(&cookie);
7091                         ANON_LOCK_EXIT(&amp->a_rwlock);
7092                 } else {
7093                         vp = svd->vp;
7094                         off = svd->offset + ptob(page);
7095                 }
7096                 if (vp == NULL) {               /* untouched zfod page */
7097                         ASSERT(ap == NULL);
7098                         continue;
7099                 }

7101                 pp = page_lookup_nowait(vp, off, SE_SHARED);
7102                 if (pp == NULL)
7103                         continue;


7106                 /*
7107                  * Examine the page to see whether it can be tossed out,
7108                  * keeping track of how many we've found.
7109                  */
7110                 if (!page_tryupgrade(pp)) {
7111                         /*
7112                          * If the page has an i/o lock and no mappings,
7113                          * it's very likely that the page is being
7114                          * written out as a result of klustering.
7115                          * Assume this is so and take credit for it here.
7116                          */
7117                         if (!page_io_trylock(pp)) {
7118                                 if (!hat_page_is_mapped(pp))
7119                                         pgcnt++;
7120                         } else {
7121                                 page_io_unlock(pp);
7122                         }
```

```
7123                    page_unlock(pp);
7124                    continue;
7125            }
7126            ASSERT(!page_iolock_assert(pp));


7129            /*
7130             * Skip if page is locked or has mappings.
7131             * We don't need the page_struct_lock to look at lckcnt
7132             * and cowcnt because the page is exclusive locked.
7133             */
7134            if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0 ||
7135                hat_page_is_mapped(pp)) {
7136                    page_unlock(pp);
7137                    continue;
7138            }

7140            /*
7141             * dispose skips large pages so try to demote first.
7142             */
7143            if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
7144                    page_unlock(pp);
7145                    /*
7146                     * XXX should skip the remaining page_t's of this
7147                     * large page.
7148                     */
7149                    continue;
7150            }

7152            ASSERT(pp->p_szc == 0);

7154            /*
7155             * No longer mapped -- we can toss it out.  How
7156             * we do so depends on whether or not it's dirty.
7157             */
7158            if (hat_ismod(pp) && pp->p_vnode) {
7159                    /*
7160                     * We must clean the page before it can be
7161                     * freed.  Setting B_FREE will cause pvn_done
7162                     * to free the page when the i/o completes.
7163                     * XXX: This also causes it to be accounted
7164                     *      as a pageout instead of a swap: need
7165                     *      B_SWAPOUT bit to use instead of B_FREE.
7166                     *
7167                     * Hold the vnode before releasing the page lock
7168                     * to prevent it from being freed and re-used by
7169                     * some other thread.
7170                     */
7171                    VN_HOLD(vp);
7172                    page_unlock(pp);

7174                    /*
7175                     * Queue all i/o requests for the pageout thread
7176                     * to avoid saturating the pageout devices.
7177                     */
7178                    if (!queue_io_request(vp, off))
7179                            VN_RELE(vp);
7180            } else {
7181                    /*
7182                     * The page was clean, free it.
7183                     *
7184                     * XXX: Can we ever encounter modified pages
7185                     *      with no associated vnode here?
7186                     */
7187                    ASSERT(pp->p_vnode != NULL);
7188                    /*LINTED: constant in conditional context*/
```



```
7189                            VN_DISPOSE(pp, B_FREE, 0, kcred);
7190                    }

7192                    /*
7193                     * Credit now even if i/o is in progress.
7194                     */
7195                    pgcnt++;
7196            }
7197            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

7199            /*
7200             * Wakeup pageout to initiate i/o on all queued requests.
7201             */
7202            cv_signal_pageout();
7203            return (ptob(pgcnt));
7204    }

7206    /*
7207     * Synchronize primary storage cache with real object in virtual memory.
7208     *
7209     * XXX - Anonymous pages should not be sync'ed out at all.
7210     */
7211    static int
7212    segvn_sync(struct seg *seg, caddr_t addr, size_t len, int attr, uint_t flags)
7213    {
7214            struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7215            struct vpage *vpp;
7216            page_t *pp;
7217            u_offset_t offset;
7218            struct vnode *vp;
7219            u_offset_t off;
7220            caddr_t eaddr;
7221            int bflags;
7222            int err = 0;
7223            int segtype;
7224            int pageprot;
7225            int prot;
7226            ulong_t anon_index;
7227            struct anon_map *amp;
7228            struct anon *ap;
7229            anon_sync_obj_t cookie;

7231            ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7233            SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

7235            if (svd->softlockcnt > 0) {
7236                    /*
7237                     * If this is shared segment non 0 softlockcnt
7238                     * means locked pages are still in use.
7239                     */
7240                    if (svd->type == MAP_SHARED) {
7241                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7242                            return (EAGAIN);
7243                    }

7245                    /*
7246                     * flush all pages from seg cache
7247                     * otherwise we may deadlock in swap_putpage
7248                     * for B_INVAL page (4175402).
7249                     *
7250                     * Even if we grab segvn WRITER's lock
7251                     * here, there might be another thread which could've
7252                     * successfully performed lookup/insert just before
7253                     * we acquired the lock here.  So, grabbing either
7254                     * lock here is of not much use.  Until we devise
```

```
7255                 * a strategy at upper layers to solve the
7256                 * synchronization issues completely, we expect
7257                 * applications to handle this appropriately.
7258                 */
7259                segvn_purge(seg);
7260                if (svd->softlockcnt > 0) {
7261                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7262                        return (EAGAIN);
7263                }
7264        } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
7265            svd->amp->a_softlockcnt > 0) {
7266                /*
7267                 * Try to purge this amp's entries from pcache. It will
7268                 * succeed only if other segments that share the amp have no
7269                 * outstanding softlock's.
7270                 */
7271                segvn_purge(seg);
7272                if (svd->amp->a_softlockcnt > 0 || svd->softlockcnt > 0) {
7273                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7274                        return (EAGAIN);
7275                }
7276        }

7278        vpp = svd->vpage;
7279        offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7280        bflags = ((flags & MS_ASYNC) ? B_ASYNC : 0) |
7281            ((flags & MS_INVALIDATE) ? B_INVAL : 0);

7283        if (attr) {
7284                pageprot = attr & ~(SHARED|PRIVATE);
7285                segtype = (attr & SHARED) ? MAP_SHARED : MAP_PRIVATE;

7287                /*
7288                 * We are done if the segment types don't match
7289                 * or if we have segment level protections and
7290                 * they don't match.
7291                 */
7292                if (svd->type != segtype) {
7293                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7294                        return (0);
7295                }
7296                if (vpp == NULL) {
7297                        if (svd->prot != pageprot) {
7298                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7299                                return (0);
7300                        }
7301                        prot = svd->prot;
7302                } else
7303                        vpp = &svd->vpage[seg_page(seg, addr)];

7305        } else if (svd->vp && svd->amp == NULL &&
7306            (flags & MS_INVALIDATE) == 0) {

7308                /*
7309                 * No attributes, no anonymous pages and MS_INVALIDATE flag
7310                 * is not on, just use one big request.
7311                 */
7312                err = VOP_PUTPAGE(svd->vp, (offset_t)offset, len,
7313                    bflags, svd->cred, NULL);
7314                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7315                return (err);
7316        }

7318        if ((amp = svd->amp) != NULL)
7319                anon_index = svd->anon_index + seg_page(seg, addr);
```



```
7321        for (eaddr = addr + len; addr < eaddr; addr += PAGESIZE) {
7322                ap = NULL;
7323                if (amp != NULL) {
7324                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7325                        anon_array_enter(amp, anon_index, &cookie);
7326                        ap = anon_get_ptr(amp->ahp, anon_index++);
7327                        if (ap != NULL) {
7328                                swap_xlate(ap, &vp, &off);
7329                        } else {
7330                                vp = svd->vp;
7331                                off = offset;
7332                        }
7333                        anon_array_exit(&cookie);
7334                        ANON_LOCK_EXIT(&amp->a_rwlock);
7335                } else {
7336                        vp = svd->vp;
7337                        off = offset;
7338                }
7339                offset += PAGESIZE;

7341                if (vp == NULL)         /* untouched zfod page */
7342                        continue;

7344                if (attr) {
7345                        if (vpp) {
7346                                prot = VPP_PROT(vpp);
7347                                vpp++;
7348                        }
7349                        if (prot != pageprot) {
7350                                continue;
7351                        }
7352                }

7354                /*
7355                 * See if any of these pages are locked --  if so, then we
7356                 * will have to truncate an invalidate request at the first
7357                 * locked one. We don't need the page_struct_lock to test
7358                 * as this is only advisory; even if we acquire it someone
7359                 * might race in and lock the page after we unlock and before
7360                 * we do the PUTPAGE, then PUTPAGE simply does nothing.
7361                 */
7362                if (flags & MS_INVALIDATE) {
7363                        if ((pp = page_lookup(vp, off, SE_SHARED)) != NULL) {
7364                                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0) {
7365                                        page_unlock(pp);
7366                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7367                                        return (EBUSY);
7368                                }
7369                                if (ap != NULL && pp->p_szc != 0 &&
7370                                    page_tryupgrade(pp)) {
7371                                        if (pp->p_lckcnt == 0 &&
7372                                            pp->p_cowcnt == 0) {
7373                                                /*
7374                                                 * swapfs VN_DISPOSE() won't
7375                                                 * invalidate large pages.
7376                                                 * Attempt to demote.
7377                                                 * XXX can't help it if it
7378                                                 * fails. But for swapfs
7379                                                 * pages it is no big deal.
7380                                                 */
7381                                                (void) page_try_demote_pages(
7382                                                    pp);
7383                                        }
7384                                }
7385                                page_unlock(pp);
7386                        }
```

```
7387                } else if (svd->type == MAP_SHARED && amp != NULL) {
7388                        /*
7389                         * Avoid writing out to disk ISM's large pages
7390                         * because segspt_free_pages() relies on NULL an_pvp
7391                         * of anon slots of such pages.
7392                         */
7394                        ASSERT(svd->vp == NULL);
7395                        /*
7396                         * swapfs uses page_lookup_nowait if not freeing or
7397                         * invalidating and skips a page if
7398                         * page_lookup_nowait returns NULL.
7399                         */
7400                        pp = page_lookup_nowait(vp, off, SE_SHARED);
7401                        if (pp == NULL) {
7402                                continue;
7403                        }
7404                        if (pp->p_szc != 0) {
7405                                page_unlock(pp);
7406                                continue;
7407                        }
7409                        /*
7410                         * Note ISM pages are created large so (vp, off)'s
7411                         * page cannot suddenly become large after we unlock
7412                         * pp.
7413                         */
7414                        page_unlock(pp);
7415                }
7416                /*
7417                 * XXX - Should ultimately try to kluster
7418                 * calls to VOP_PUTPAGE() for performance.
7419                 */
7420                VN_HOLD(vp);
7421                err = VOP_PUTPAGE(vp, (offset_t)off, PAGESIZE,
7422                    (bflags | (IS_SWAPFSVP(vp) ? B_PAGE_NOWAIT : 0)),
7423                    svd->cred, NULL);
7425                VN_RELE(vp);
7426                if (err)
7427                        break;
7428        }
7429        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7430        return (err);
7431 }
7433 /*
7434  * Determine if we have data corresponding to pages in the
7435  * primary storage virtual memory cache (i.e., "in core").
7436  */
7437 static size_t
7438 segvn_incore(struct seg *seg, caddr_t addr, size_t len, char *vec)
7439 {
7440        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7441        struct vnode *vp, *avp;
7442        u_offset_t offset, aoffset;
7443        size_t p, ep;
7444        int ret;
7445        struct vpage *vpp;
7446        page_t *pp;
7447        uint_t start;
7448        struct anon_map *amp;           /* XXX - for locknest */
7449        struct anon *ap;
7450        uint_t attr;
7451        anon_sync_obj_t cookie;
```



```
7453        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
7455        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
7456        if (svd->amp == NULL && svd->vp == NULL) {
7457                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7458                bzero(vec, btopr(len));
7459                return (len);   /* no anonymous pages created yet */
7460        }
7462        p = seg_page(seg, addr);
7463        ep = seg_page(seg, addr + len);
7464        start = svd->vp ? SEG_PAGE_VNODEBACKED : 0;
7466        amp = svd->amp;
7467        for (; p < ep; p++, addr += PAGESIZE) {
7468                vpp = (svd->vpage) ? &svd->vpage[p]: NULL;
7469                ret = start;
7470                ap = NULL;
7471                avp = NULL;
7472                /* Grab the vnode/offset for the anon slot */
7473                if (amp != NULL) {
7474                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7475                        anon_array_enter(amp, svd->anon_index + p, &cookie);
7476                        ap = anon_get_ptr(amp->ahp, svd->anon_index + p);
7477                        if (ap != NULL) {
7478                                swap_xlate(ap, &avp, &aoffset);
7479                        }
7480                        anon_array_exit(&cookie);
7481                        ANON_LOCK_EXIT(&amp->a_rwlock);
7482                }
7483                if ((avp != NULL) && page_exists(avp, aoffset)) {
7484                        /* A page exists for the anon slot */
7485                        ret |= SEG_PAGE_INCORE;
7487                        /*
7488                         * If page is mapped and writable
7489                         */
7490                        attr = (uint_t)0;
7491                        if ((hat_getattr(seg->s_as->a_hat, addr,
7492                            &attr) != -1) && (attr & PROT_WRITE)) {
7493                                ret |= SEG_PAGE_ANON;
7494                        }
7495                        /*
7496                         * Don't get page_struct lock for lckcnt and cowcnt,
7497                         * since this is purely advisory.
7498                         */
7499                        if ((pp = page_lookup_nowait(avp, aoffset,
7500                            SE_SHARED)) != NULL) {
7501                                if (pp->p_lckcnt)
7502                                        ret |= SEG_PAGE_SOFTLOCK;
7503                                if (pp->p_cowcnt)
7504                                        ret |= SEG_PAGE_HASCOW;
7505                                page_unlock(pp);
7506                        }
7507                }
7509                /* Gather vnode statistics */
7510                vp = svd->vp;
7511                offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7513                if (vp != NULL) {
7514                        /*
7515                         * Try to obtain a "shared" lock on the page
7516                         * without blocking.  If this fails, determine
7517                         * if the page is in memory.
7518                         */
```

```
7519                        pp = page_lookup_nowait(vp, offset, SE_SHARED);
7520                        if ((pp == NULL) && (page_exists(vp, offset))) {
7521                                /* Page is incore, and is named */
7522                                ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
7523                        }
7524                        /*
7525                         * Don't get page_struct lock for lckcnt and cowcnt,
7526                         * since this is purely advisory.
7527                         */
7528                        if (pp != NULL) {
7529                                ret |= (SEG_PAGE_INCORE | SEG_PAGE_VNODE);
7530                                if (pp->p_lckcnt)
7531                                        ret |= SEG_PAGE_SOFTLOCK;
7532                                if (pp->p_cowcnt)
7533                                        ret |= SEG_PAGE_HASCOW;
7534                                page_unlock(pp);
7535                        }
7536                }

7538                /* Gather virtual page information */
7539                if (vpp) {
7540                        if (VPP_ISPPLOCK(vpp))
7541                                ret |= SEG_PAGE_LOCKED;
7542                        vpp++;
7543                }

7545                *vec++ = (char)ret;
7546        }
7547        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7548        return (len);
7549 }

7551 /*
7552  * Statement for p_cowcnts/p_lckcnts.
7553  *
7554  * p_cowcnt is updated while mlock/munlocking MAP_PRIVATE and PROT_WRITE region
7555  * irrespective of the following factors or anything else:
7556  *
7557  *      (1) anon slots are populated or not
7558  *      (2) cow is broken or not
7559  *      (3) refcnt on ap is 1 or greater than 1
7560  *
7561  * If it's not MAP_PRIVATE and PROT_WRITE, p_lckcnt is updated during mlock
7562  * and munlock.
7563  *
7564  *
7565  * Handling p_cowcnts/p_lckcnts during copy-on-write fault:
7566  *
7567  *      if vpage has PROT_WRITE
7568  *              transfer cowcnt on the oldpage -> cowcnt on the newpage
7569  *      else
7570  *              transfer lckcnt on the oldpage -> lckcnt on the newpage
7571  *
7572  *      During copy-on-write, decrement p_cowcnt on the oldpage and increment
7573  *      p_cowcnt on the newpage *if* the corresponding vpage has PROT_WRITE.
7574  *
7575  *      We may also break COW if softlocking on read access in the physio case.
7576  *      In this case, vpage may not have PROT_WRITE. So, we need to decrement
7577  *      p_lckcnt on the oldpage and increment p_lckcnt on the newpage *if* the
7578  *      vpage doesn't have PROT_WRITE.
7579  *
7580  *
7581  * Handling p_cowcnts/p_lckcnts during mprotect on mlocked region:
7582  *
7583  *      If a MAP_PRIVATE region loses PROT_WRITE, we decrement p_cowcnt and
7584  *      increment p_lckcnt by calling page_subclaim() which takes care of
```



```
7585  *      availrmem accounting and p_lckcnt overflow.
7586  *
7587  *      If a MAP_PRIVATE region gains PROT_WRITE, we decrement p_lckcnt and
7588  *      increment p_cowcnt by calling page_addclaim() which takes care of
7589  *      availrmem availability and p_cowcnt overflow.
7590  */

7592 /*
7593  * Lock down (or unlock) pages mapped by this segment.
7594  *
7595  * XXX only creates PAGESIZE pages if anon slots are not initialized.
7596  * At fault time they will be relocated into larger pages.
7597  */
7598 static int
7599 segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
7600     int attr, int op, ulong_t *lockmap, size_t pos)
7601 {
7602        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7603        struct vpage *vpp;
7604        struct vpage *evp;
7605        page_t *pp;
7606        u_offset_t offset;
7607        u_offset_t off;
7608        int segtype;
7609        int pageprot;
7610        int claim;
7611        struct vnode *vp;
7612        ulong_t anon_index;
7613        struct anon_map *amp;
7614        struct anon *ap;
7615        struct vattr va;
7616        anon_sync_obj_t cookie;
7617        struct kshmid *sp = NULL;
7618        struct proc     *p = curproc;
7619        kproject_t      *proj = NULL;
7620        int chargeproc = 1;
7621        size_t locked_bytes = 0;
7622        size_t unlocked_bytes = 0;
7623        int err = 0;

7625        /*
7626         * Hold write lock on address space because may split or concatenate
7627         * segments
7628         */
7629        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7631        /*
7632         * If this is a shm, use shm's project and zone, else use
7633         * project and zone of calling process
7634         */

7636        /* Determine if this segment backs a sysV shm */
7637        if (svd->amp != NULL && svd->amp->a_sp != NULL) {
7638                ASSERT(svd->type == MAP_SHARED);
7639                ASSERT(svd->tr_state == SEGVN_TR_OFF);
7640                sp = svd->amp->a_sp;
7641                proj = sp->shm_perm.ipc_proj;
7642                chargeproc = 0;
7643        }

7645        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
7646        if (attr) {
7647                pageprot = attr & ~(SHARED|PRIVATE);
7648                segtype = attr & SHARED ? MAP_SHARED : MAP_PRIVATE;

7650                /*
```

```
7651                 * We are done if the segment types don't match
7652                 * or if we have segment level protections and
7653                 * they don't match.
7654                 */
7655                if (svd->type != segtype) {
7656                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7657                        return (0);
7658                }
7659                if (svd->pageprot == 0 && svd->prot != pageprot) {
7660                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7661                        return (0);
7662                }
7663        }

7665        if (op == MC_LOCK) {
7666                if (svd->tr_state == SEGVN_TR_INIT) {
7667                        svd->tr_state = SEGVN_TR_OFF;
7668                } else if (svd->tr_state == SEGVN_TR_ON) {
7669                        ASSERT(svd->amp != NULL);
7670                        segvn_textunrepl(seg, 0);
7671                        ASSERT(svd->amp == NULL &&
7672                            svd->tr_state == SEGVN_TR_OFF);
7673                }
7674        }

7676        /*
7677         * If we're locking, then we must create a vpage structure if
7678         * none exists.  If we're unlocking, then check to see if there
7679         * is a vpage --  if not, then we could not have locked anything.
7680         */

7682        if ((vpp = svd->vpage) == NULL) {
7683                if (op == MC_LOCK)
7684                        segvn_vpage(seg);
7685                else {
7686                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7687                        return (0);
7688                }
7689        }

7691        /*
7692         * The anonymous data vector (i.e., previously
7693         * unreferenced mapping to swap space) can be allocated
7694         * by lazily testing for its existence.
7695         */
7696        if (op == MC_LOCK && svd->amp == NULL && svd->vp == NULL) {
7697                ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
7698                svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
7699                svd->amp->a_szc = seg->s_szc;
7700        }

7702        if ((amp = svd->amp) != NULL) {
7703                anon_index = svd->anon_index + seg_page(seg, addr);
7704        }

7706        offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7707        evp = &svd->vpage[seg_page(seg, addr + len)];

7709        if (sp != NULL)
7710                mutex_enter(&sp->shm_mlock);

7712        /* determine number of unlocked bytes in range for lock operation */
7713        if (op == MC_LOCK) {

7715                if (sp == NULL) {
7716                        for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
```



```
7717                            vpp++) {
7718                                if (!VPP_ISPPLOCK(vpp))
7719                                        unlocked_bytes += PAGESIZE;
7720                        }
7721                } else {
7722                        ulong_t         i_idx, i_edx;
7723                        anon_sync_obj_t i_cookie;
7724                        struct anon     *i_ap;
7725                        struct vnode    *i_vp;
7726                        u_offset_t      i_off;

7728                        /* Only count sysV pages once for locked memory */
7729                        i_edx = svd->anon_index + seg_page(seg, addr + len);
7730                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7731                        for (i_idx = anon_index; i_idx < i_edx; i_idx++) {
7732                                anon_array_enter(amp, i_idx, &i_cookie);
7733                                i_ap = anon_get_ptr(amp->ahp, i_idx);
7734                                if (i_ap == NULL) {
7735                                        unlocked_bytes += PAGESIZE;
7736                                        anon_array_exit(&i_cookie);
7737                                        continue;
7738                                }
7739                                swap_xlate(i_ap, &i_vp, &i_off);
7740                                anon_array_exit(&i_cookie);
7741                                pp = page_lookup(i_vp, i_off, SE_SHARED);
7742                                if (pp == NULL) {
7743                                        unlocked_bytes += PAGESIZE;
7744                                        continue;
7745                                } else if (pp->p_lckcnt == 0)
7746                                        unlocked_bytes += PAGESIZE;
7747                                page_unlock(pp);
7748                        }
7749                        ANON_LOCK_EXIT(&amp->a_rwlock);
7750                }

7752                mutex_enter(&p->p_lock);
7753                err = rctl_incr_locked_mem(p, proj, unlocked_bytes,
7754                    chargeproc);
7755                mutex_exit(&p->p_lock);

7757                if (err) {
7758                        if (sp != NULL)
7759                                mutex_exit(&sp->shm_mlock);
7760                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7761                        return (err);
7762                }
7763        }
7764        /*
7765         * Loop over all pages in the range.  Process if we're locking and
7766         * page has not already been locked in this mapping; or if we're
7767         * unlocking and the page has been locked.
7768         */
7769        for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
7770            vpp++, pos++, addr += PAGESIZE, offset += PAGESIZE, anon_index++) {
7771                if ((attr == 0 || VPP_PROT(vpp) == pageprot) &&
7772                    ((op == MC_LOCK && !VPP_ISPPLOCK(vpp)) ||
7773                    (op == MC_UNLOCK && VPP_ISPPLOCK(vpp)))) {

7775                        if (amp != NULL)
7776                                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7777                        /*
7778                         * If this isn't a MAP_NORESERVE segment and
7779                         * we're locking, allocate anon slots if they
7780                         * don't exist.  The page is brought in later on.
7781                         */
7782                        if (op == MC_LOCK && svd->vp == NULL &&
```

```
7783                        ((svd->flags & MAP_NORESERVE) == 0) &&
7784                        amp != NULL &&
7785                        ((ap = anon_get_ptr(amp->ahp, anon_index))
7786                        == NULL)) {
7787                            anon_array_enter(amp, anon_index, &cookie);

7789                            if ((ap = anon_get_ptr(amp->ahp,
7790                                anon_index)) == NULL) {
7791                                pp = anon_zero(seg, addr, &ap,
7792                                    svd->cred);
7793                                if (pp == NULL) {
7794                                        anon_array_exit(&cookie);
7795                                        ANON_LOCK_EXIT(&amp->a_rwlock);
7796                                        err = ENOMEM;
7797                                        goto out;
7798                                }
7799                                ASSERT(anon_get_ptr(amp->ahp,
7800                                    anon_index) == NULL);
7801                                (void) anon_set_ptr(amp->ahp,
7802                                    anon_index, ap, ANON_SLEEP);
7803                                page_unlock(pp);
7804                            }
7805                            anon_array_exit(&cookie);
7806                        }

7808                        /*
7809                         * Get name for page, accounting for
7810                         * existence of private copy.
7811                         */
7812                        ap = NULL;
7813                        if (amp != NULL) {
7814                            anon_array_enter(amp, anon_index, &cookie);
7815                            ap = anon_get_ptr(amp->ahp, anon_index);
7816                            if (ap != NULL) {
7817                                swap_xlate(ap, &vp, &off);
7818                            } else {
7819                                if (svd->vp == NULL &&
7820                                    (svd->flags & MAP_NORESERVE)) {
7821                                        anon_array_exit(&cookie);
7822                                        ANON_LOCK_EXIT(&amp->a_rwlock);
7823                                        continue;
7824                                }
7825                                vp = svd->vp;
7826                                off = offset;
7827                            }
7828                            if (op != MC_LOCK || ap == NULL) {
7829                                anon_array_exit(&cookie);
7830                                ANON_LOCK_EXIT(&amp->a_rwlock);
7831                            }
7832                        } else {
7833                            vp = svd->vp;
7834                            off = offset;
7835                        }

7837                        /*
7838                         * Get page frame.  It's ok if the page is
7839                         * not available when we're unlocking, as this
7840                         * may simply mean that a page we locked got
7841                         * truncated out of existence after we locked it.
7842                         *
7843                         * Invoke VOP_GETPAGE() to obtain the page struct
7844                         * since we may need to read it from disk if its
7845                         * been paged out.
7846                         */
7847                        if (op != MC_LOCK)
7848                            pp = page_lookup(vp, off, SE_SHARED);
```



```
7849                    else {
7850                        page_t *pl[1 + 1];
7851                        int error;

7853                        ASSERT(vp != NULL);

7855                        error = VOP_GETPAGE(vp, (offset_t)off, PAGESIZE,
7856                            (uint_t *)NULL, pl, PAGESIZE, seg, addr,
7857                            S_OTHER, svd->cred, NULL);

7859                        if (error && ap != NULL) {
7860                            anon_array_exit(&cookie);
7861                            ANON_LOCK_EXIT(&amp->a_rwlock);
7862                        }

7864                        /*
7865                         * If the error is EDEADLK then we must bounce
7866                         * up and drop all vm subsystem locks and then
7867                         * retry the operation later
7868                         * This behavior is a temporary measure because
7869                         * ufs/sds logging is badly designed and will
7870                         * deadlock if we don't allow this bounce to
7871                         * happen.  The real solution is to re-design
7872                         * the logging code to work properly.  See bug
7873                         * 4125102 for details of the problem.
7874                         */
7875                        if (error == EDEADLK) {
7876                            err = error;
7877                            goto out;
7878                        }
7879                        /*
7880                         * Quit if we fail to fault in the page.  Treat
7881                         * the failure as an error, unless the addr
7882                         * is mapped beyond the end of a file.
7883                         */
7884                        if (error && svd->vp) {
7885                            va.va_mask = AT_SIZE;
7886                            if (VOP_GETATTR(svd->vp, &va, 0,
7887                                svd->cred, NULL) != 0) {
7888                                err = EIO;
7889                                goto out;
7890                            }
7891                            if (btopr(va.va_size) >=
7892                                btopr(off + 1)) {
7893                                err = EIO;
7894                                goto out;
7895                            }
7896                            goto out;

7898                        } else if (error) {
7899                            err = EIO;
7900                            goto out;
7901                        }
7902                        pp = pl[0];
7903                        ASSERT(pp != NULL);
7904                    }

7906                    /*
7907                     * See Statement at the beginning of this routine.
7908                     *
7909                     * claim is always set if MAP_PRIVATE and PROT_WRITE
7910                     * irrespective of following factors:
7911                     *
7912                     * (1) anon slots are populated or not
7913                     * (2) cow is broken or not
7914                     * (3) refcnt on ap is 1 or greater than 1
```

```
7915                         *
7916                         * See 4140683 for details
7917                         */
7918                        claim = ((VPP_PROT(vpp) & PROT_WRITE) &&
7919                            (svd->type == MAP_PRIVATE));
7920
7921                        /*
7922                         * Perform page-level operation appropriate to
7923                         * operation.  If locking, undo the SOFTLOCK
7924                         * performed to bring the page into memory
7925                         * after setting the lock.  If unlocking,
7926                         * and no page was found, account for the claim
7927                         * separately.
7928                         */
7929                        if (op == MC_LOCK) {
7930                                int ret = 1;    /* Assume success */
7931
7932                                ASSERT(!VPP_ISPPLOCK(vpp));
7933
7934                                ret = page_pp_lock(pp, claim, 0);
7935                                if (ap != NULL) {
7936                                        if (ap->an_pvp != NULL) {
7937                                                anon_swap_free(ap, pp);
7938                                        }
7939                                        anon_array_exit(&cookie);
7940                                        ANON_LOCK_EXIT(&amp->a_rwlock);
7941                                }
7942                                if (ret == 0) {
7943                                        /* locking page failed */
7944                                        page_unlock(pp);
7945                                        err = EAGAIN;
7946                                        goto out;
7947                                }
7948                                VPP_SETPPLOCK(vpp);
7949                                if (sp != NULL) {
7950                                        if (pp->p_lckcnt == 1)
7951                                                locked_bytes += PAGESIZE;
7952                                } else
7953                                        locked_bytes += PAGESIZE;
7954
7955                                if (lockmap != (ulong_t *)NULL)
7956                                        BT_SET(lockmap, pos);
7957
7958                                page_unlock(pp);
7959                        } else {
7960                                ASSERT(VPP_ISPPLOCK(vpp));
7961                                if (pp != NULL) {
7962                                        /* sysV pages should be locked */
7963                                        ASSERT(sp == NULL || pp->p_lckcnt > 0);
7964                                        page_pp_unlock(pp, claim, 0);
7965                                        if (sp != NULL) {
7966                                                if (pp->p_lckcnt == 0)
7967                                                        unlocked_bytes
7968                                                            += PAGESIZE;
7969                                        } else
7970                                                unlocked_bytes += PAGESIZE;
7971                                        page_unlock(pp);
7972                                } else {
7973                                        ASSERT(sp == NULL);
7974                                        unlocked_bytes += PAGESIZE;
7975                                }
7976                                VPP_CLRPPLOCK(vpp);
7977                        }
7978                }
7979        }
7980 out:
```



```
7981        if (op == MC_LOCK) {
7982                /* Credit back bytes that did not get locked */
7983                if ((unlocked_bytes - locked_bytes) > 0) {
7984                        if (proj == NULL)
7985                                mutex_enter(&p->p_lock);
7986                        rctl_decr_locked_mem(p, proj,
7987                            (unlocked_bytes - locked_bytes), chargeproc);
7988                        if (proj == NULL)
7989                                mutex_exit(&p->p_lock);
7990                }
7991
7992        } else {
7993                /* Account bytes that were unlocked */
7994                if (unlocked_bytes > 0) {
7995                        if (proj == NULL)
7996                                mutex_enter(&p->p_lock);
7997                        rctl_decr_locked_mem(p, proj, unlocked_bytes,
7998                            chargeproc);
7999                        if (proj == NULL)
8000                                mutex_exit(&p->p_lock);
8001                }
8002        }
8003        if (sp != NULL)
8004                mutex_exit(&sp->shm_mlock);
8005        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8006
8007        return (err);
8008 }
8009
8010 /*
8011  * Set advice from user for specified pages
8012  * There are 5 types of advice:
8013  *      MADV_NORMAL     - Normal (default) behavior (whatever that is)
8014  *      MADV_RANDOM     - Random page references
8015  *                              do not allow readahead or 'klustering'
8016  *      MADV_SEQUENTIAL - Sequential page references
8017  *                              Pages previous to the one currently being
8018  *                              accessed (determined by fault) are 'not needed'
8019  *                              and are freed immediately
8020  *      MADV_WILLNEED   - Pages are likely to be used (fault ahead in mctl)
8021  *      MADV_DONTNEED   - Pages are not needed (synced out in mctl)
8022  *      MADV_FREE       - Contents can be discarded
8023  *      MADV_ACCESS_DEFAULT- Default access
8024  *      MADV_ACCESS_LWP - Next LWP will access heavily
8025  *      MADV_ACCESS_MANY- Many LWPs or processes will access heavily
8026  */
8027 static int
8028 segvn_advise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
8029 {
8030        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8031        size_t page;
8032        int err = 0;
8033        int already_set;
8034        struct anon_map *amp;
8035        ulong_t anon_index;
8036        struct seg *next;
8037        lgrp_mem_policy_t policy;
8038        struct seg *prev;
8039        struct vnode *vp;
8040
8041        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
8042
8043        /*
8044         * In case of MADV_FREE, we won't be modifying any segment private
8045         * data structures; so, we only need to grab READER's lock
8046         */
```

```
8047         if (behav != MADV_FREE) {
8048                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
8049                 if (svd->tr_state != SEGVN_TR_OFF) {
8050                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8051                         return (0);
8052                 }
8053         } else {
8054                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
8055         }

8057         /*
8058          * Large pages are assumed to be only turned on when accesses to the
8059          * segment's address range have spatial and temporal locality. That
8060          * justifies ignoring MADV_SEQUENTIAL for large page segments.
8061          * Also, ignore advice affecting lgroup memory allocation
8062          * if don't need to do lgroup optimizations on this system
8063          */

8065         if ((behav == MADV_SEQUENTIAL &&
8066             (seg->s_szc != 0 || HAT_IS_REGION_COOKIE_VALID(svd->rcookie))) ||
8067             (!lgrp_optimizations() && (behav == MADV_ACCESS_DEFAULT ||
8068             behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY))) {
8069                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8070                 return (0);
8071         }

8073         if (behav == MADV_SEQUENTIAL || behav == MADV_ACCESS_DEFAULT ||
8074             behav == MADV_ACCESS_LWP || behav == MADV_ACCESS_MANY) {
8075                 /*
8076                  * Since we are going to unload hat mappings
8077                  * we first have to flush the cache. Otherwise
8078                  * this might lead to system panic if another
8079                  * thread is doing physio on the range whose
8080                  * mappings are unloaded by madvise(3C).
8081                  */
8082                 if (svd->softlockcnt > 0) {
8083                         /*
8084                          * If this is shared segment non 0 softlockcnt
8085                          * means locked pages are still in use.
8086                          */
8087                         if (svd->type == MAP_SHARED) {
8088                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8089                                 return (EAGAIN);
8090                         }
8091                         /*
8092                          * Since we do have the segvn writers lock
8093                          * nobody can fill the cache with entries
8094                          * belonging to this seg during the purge.
8095                          * The flush either succeeds or we still
8096                          * have pending I/Os. In the later case,
8097                          * madvise(3C) fails.
8098                          */
8099                         segvn_purge(seg);
8100                         if (svd->softlockcnt > 0) {
8101                                 /*
8102                                  * Since madvise(3C) is advisory and
8103                                  * it's not part of UNIX98, madvise(3C)
8104                                  * failure here doesn't cause any hardship.
8105                                  * Note that we don't block in "as" layer.
8106                                  */
8107                                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8108                                 return (EAGAIN);
8109                         }
8110                 } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
8111                     svd->amp->a_softlockcnt > 0) {
8112                         /*
```



```
8113                          * Try to purge this amp's entries from pcache. It
8114                          * will succeed only if other segments that share the
8115                          * amp have no outstanding softlock's.
8116                          */
8117                         segvn_purge(seg);
8118                 }
8119         }

8121         amp = svd->amp;
8122         vp = svd->vp;
8123         if (behav == MADV_FREE) {
8124                 /*
8125                  * MADV_FREE is not supported for segments with
8126                  * underlying object; if anonmap is NULL, anon slots
8127                  * are not yet populated and there is nothing for
8128                  * us to do. As MADV_FREE is advisory, we don't
8129                  * return error in either case.
8130                  */
8131                 if (vp != NULL || amp == NULL) {
8132                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8133                         return (0);
8134                 }

8136                 segvn_purge(seg);

8138                 page = seg_page(seg, addr);
8139                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
8140                 anon_disclaim(amp, svd->anon_index + page, len);
8141                 ANON_LOCK_EXIT(&amp->a_rwlock);
8142                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8143                 return (0);
8144         }

8146         /*
8147          * If advice is to be applied to entire segment,
8148          * use advice field in seg_data structure
8149          * otherwise use appropriate vpage entry.
8150          */
8151         if ((addr == seg->s_base) && (len == seg->s_size)) {
8152                 switch (behav) {
8153                 case MADV_ACCESS_LWP:
8154                 case MADV_ACCESS_MANY:
8155                 case MADV_ACCESS_DEFAULT:
8156                         /*
8157                          * Set memory allocation policy for this segment
8158                          */
8159                         policy = lgrp_madv_to_policy(behav, len, svd->type);
8160                         if (svd->type == MAP_SHARED)
8161                                 already_set = lgrp_shm_policy_set(policy, amp,
8162                                     svd->anon_index, vp, svd->offset, len);
8163                         else {
8164                                 /*
8165                                  * For private memory, need writers lock on
8166                                  * address space because the segment may be
8167                                  * split or concatenated when changing policy
8168                                  */
8169                                 if (AS_READ_HELD(seg->s_as,
8170                                     &seg->s_as->a_lock)) {
8171                                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8172                                         return (IE_RETRY);
8173                                 }

8175                                 already_set = lgrp_privm_policy_set(policy,
8176                                     &svd->policy_info, len);
8177                         }
```

```
8179                        /*
8180                         * If policy set already and it shouldn't be reapplied,
8181                         * don't do anything.
8182                         */
8183                        if (already_set &&
8184                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
8185                                break;
8186
8187                        /*
8188                         * Mark any existing pages in given range for
8189                         * migration
8190                         */
8191                        page_mark_migrate(seg, addr, len, amp, svd->anon_index,
8192                            vp, svd->offset, 1);
8193
8194                        /*
8195                         * If same policy set already or this is a shared
8196                         * memory segment, don't need to try to concatenate
8197                         * segment with adjacent ones.
8198                         */
8199                        if (already_set || svd->type == MAP_SHARED)
8200                                break;
8201
8202                        /*
8203                         * Try to concatenate this segment with previous
8204                         * one and next one, since we changed policy for
8205                         * this one and it may be compatible with adjacent
8206                         * ones now.
8207                         */
8208                        prev = AS_SEGPREV(seg->s_as, seg);
8209                        next = AS_SEGNEXT(seg->s_as, seg);
8210
8211                        if (next && next->s_ops == &segvn_ops &&
8212                            addr + len == next->s_base)
8213                                (void) segvn_concat(seg, next, 1);
8214
8215                        if (prev && prev->s_ops == &segvn_ops &&
8216                            addr == prev->s_base + prev->s_size) {
8217                                /*
8218                                 * Drop lock for private data of current
8219                                 * segment before concatenating (deleting) it
8220                                 * and return IE_REATTACH to tell as_ctl() that
8221                                 * current segment has changed
8222                                 */
8223                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8224                                if (!segvn_concat(prev, seg, 1))
8225                                        err = IE_REATTACH;
8226
8227                                return (err);
8228                        }
8229                        break;
8230
8231                case MADV_SEQUENTIAL:
8232                        /*
8233                         * unloading mapping guarantees
8234                         * detection in segvn_fault
8235                         */
8236                        ASSERT(seg->s_szc == 0);
8237                        ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
8238                        hat_unload(seg->s_as->a_hat, addr, len,
8239                            HAT_UNLOAD);
8240                        /* FALLTHROUGH */
8241                case MADV_NORMAL:
8242                case MADV_RANDOM:
8243                        svd->advice = (uchar_t)behav;
8244                        svd->pageadvice = 0;
```



```
8245                        break;
8246                case MADV_WILLNEED:     /* handled in memcntl */
8247                case MADV_DONTNEED:     /* handled in memcntl */
8248                case MADV_FREE:         /* handled above */
8249                        break;
8250                default:
8251                        err = EINVAL;
8252                }
8253        } else {
8254                caddr_t                 eaddr;
8255                struct seg              *new_seg;
8256                struct segvn_data       *new_svd;
8257                u_offset_t              off;
8258                caddr_t                 oldeaddr;
8259
8260                page = seg_page(seg, addr);
8261
8262                segvn_vpage(seg);
8263
8264                switch (behav) {
8265                        struct vpage *bvpp, *evpp;
8266
8267                case MADV_ACCESS_LWP:
8268                case MADV_ACCESS_MANY:
8269                case MADV_ACCESS_DEFAULT:
8270                        /*
8271                         * Set memory allocation policy for portion of this
8272                         * segment
8273                         */
8274
8275                        /*
8276                         * Align address and length of advice to page
8277                         * boundaries for large pages
8278                         */
8279                        if (seg->s_szc != 0) {
8280                                size_t  pgsz;
8281
8282                                pgsz = page_get_pagesize(seg->s_szc);
8283                                addr = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
8284                                len = P2ROUNDUP(len, pgsz);
8285                        }
8286
8287                        /*
8288                         * Check to see whether policy is set already
8289                         */
8290                        policy = lgrp_madv_to_policy(behav, len, svd->type);
8291
8292                        anon_index = svd->anon_index + page;
8293                        off = svd->offset + (uintptr_t)(addr - seg->s_base);
8294
8295                        if (svd->type == MAP_SHARED)
8296                                already_set = lgrp_shm_policy_set(policy, amp,
8297                                    anon_index, vp, off, len);
8298                        else
8299                                already_set =
8300                                    (policy == svd->policy_info.mem_policy);
8301
8302                        /*
8303                         * If policy set already and it shouldn't be reapplied,
8304                         * don't do anything.
8305                         */
8306                        if (already_set &&
8307                            !LGRP_MEM_POLICY_REAPPLICABLE(policy))
8308                                break;
8309
8310                        /*
```

```
8311                     * For private memory, need writers lock on
8312                     * address space because the segment may be
8313                     * split or concatenated when changing policy
8314                     */
8315                    if (svd->type == MAP_PRIVATE &&
8316                        AS_READ_HELD(seg->s_as, &seg->s_as->a_lock)) {
8317                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8318                            return (IE_RETRY);
8319                    }

8321                    /*
8322                     * Mark any existing pages in given range for
8323                     * migration
8324                     */
8325                    page_mark_migrate(seg, addr, len, amp, svd->anon_index,
8326                        vp, svd->offset, 1);

8328                    /*
8329                     * Don't need to try to split or concatenate
8330                     * segments, since policy is same or this is a shared
8331                     * memory segment
8332                     */
8333                    if (already_set || svd->type == MAP_SHARED)
8334                            break;

8336                    if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
8337                            ASSERT(svd->amp == NULL);
8338                            ASSERT(svd->tr_state == SEGVN_TR_OFF);
8339                            ASSERT(svd->softlockcnt == 0);
8340                            hat_leave_region(seg->s_as->a_hat, svd->rcookie,
8341                                HAT_REGION_TEXT);
8342                            svd->rcookie = HAT_INVALID_REGION_COOKIE;
8343                    }

8345                    /*
8346                     * Split off new segment if advice only applies to a
8347                     * portion of existing segment starting in middle
8348                     */
8349                    new_seg = NULL;
8350                    eaddr = addr + len;
8351                    oldeaddr = seg->s_base + seg->s_size;
8352                    if (addr > seg->s_base) {
8353                            /*
8354                             * Must flush I/O page cache
8355                             * before splitting segment
8356                             */
8357                            if (svd->softlockcnt > 0)
8358                                    segvn_purge(seg);

8360                            /*
8361                             * Split segment and return IE_REATTACH to tell
8362                             * as_ctl() that current segment changed
8363                             */
8364                            new_seg = segvn_split_seg(seg, addr);
8365                            new_svd = (struct segvn_data *)new_seg->s_data;
8366                            err = IE_REATTACH;

8368                            /*
8369                             * If new segment ends where old one
8370                             * did, try to concatenate the new
8371                             * segment with next one.
8372                             */
8373                            if (eaddr == oldeaddr) {
8374                                    /*
8375                                     * Set policy for new segment
8376                                     */
```



```
8377                                    (void) lgrp_privm_policy_set(policy,
8378                                        &new_svd->policy_info,
8379                                        new_seg->s_size);

8381                                    next = AS_SEGNEXT(new_seg->s_as,
8382                                        new_seg);

8384                                    if (next &&
8385                                        next->s_ops == &segvn_ops &&
8386                                        eaddr == next->s_base)
8387                                            (void) segvn_concat(new_seg,
8388                                                next, 1);
8389                            }
8390                    }

8392                    /*
8393                     * Split off end of existing segment if advice only
8394                     * applies to a portion of segment ending before
8395                     * end of the existing segment
8396                     */
8397                    if (eaddr < oldeaddr) {
8398                            /*
8399                             * Must flush I/O page cache
8400                             * before splitting segment
8401                             */
8402                            if (svd->softlockcnt > 0)
8403                                    segvn_purge(seg);

8405                            /*
8406                             * If beginning of old segment was already
8407                             * split off, use new segment to split end off
8408                             * from.
8409                             */
8410                            if (new_seg != NULL && new_seg != seg) {
8411                                    /*
8412                                     * Split segment
8413                                     */
8414                                    (void) segvn_split_seg(new_seg, eaddr);

8416                                    /*
8417                                     * Set policy for new segment
8418                                     */
8419                                    (void) lgrp_privm_policy_set(policy,
8420                                        &new_svd->policy_info,
8421                                        new_seg->s_size);
8422                            } else {
8423                                    /*
8424                                     * Split segment and return IE_REATTACH
8425                                     * to tell as_ctl() that current
8426                                     * segment changed
8427                                     */
8428                                    (void) segvn_split_seg(seg, eaddr);
8429                                    err = IE_REATTACH;

8431                                    (void) lgrp_privm_policy_set(policy,
8432                                        &svd->policy_info, seg->s_size);

8434                                    /*
8435                                     * If new segment starts where old one
8436                                     * did, try to concatenate it with
8437                                     * previous segment.
8438                                     */
8439                                    if (addr == seg->s_base) {
8440                                            prev = AS_SEGPREV(seg->s_as,
8441                                                seg);
```

```
8443                                            /*
8444                                             * Drop lock for private data
8445                                             * of current segment before
8446                                             * concatenating (deleting) it
8447                                             */
8448                                            if (prev &&
8449                                                prev->s_ops ==
8450                                                &segvn_ops &&
8451                                                addr == prev->s_base +
8452                                                prev->s_size) {
8453                                                    SEGVN_LOCK_EXIT(
8454                                                        seg->s_as,
8455                                                        &svd->lock);
8456                                                    (void) segvn_concat(
8457                                                        prev, seg, 1);
8458                                                    return (err);
8459                                            }
8460                                    }
8461                            }
8462                    }
8463                    break;
8464            case MADV_SEQUENTIAL:
8465                    ASSERT(seg->s_szc == 0);
8466                    ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
8467                    hat_unload(seg->s_as->a_hat, addr, len, HAT_UNLOAD);
8468                    /* FALLTHROUGH */
8469            case MADV_NORMAL:
8470            case MADV_RANDOM:
8471                    bvpp = &svd->vpage[page];
8472                    evpp = &svd->vpage[page + (len >> PAGESHIFT)];
8473                    for (; bvpp < evpp; bvpp++)
8474                            VPP_SETADVICE(bvpp, behav);
8475                    svd->advice = MADV_NORMAL;
8476                    break;
8477            case MADV_WILLNEED:     /* handled in memcntl */
8478            case MADV_DONTNEED:     /* handled in memcntl */
8479            case MADV_FREE:         /* handled above */
8480                    break;
8481            default:
8482                    err = EINVAL;
8483            }
8484    }
8485    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8486    return (err);
8487 }

8489 /*
8490  * Create a vpage structure for this seg.
8491  */
8492 static void
8493 segvn_vpage(struct seg *seg)
8494 {
8495    struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8496    struct vpage *vp, *evp;

8498    ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));

8500    /*
8501     * If no vpage structure exists, allocate one.  Copy the protections
8502     * and the advice from the segment itself to the individual pages.
8503     */
8504    if (svd->vpage == NULL) {
8505            svd->pageadvice = 1;
8506            svd->vpage = kmem_zalloc(seg_pages(seg) * sizeof (struct vpage),
8507                KM_SLEEP);
8508            evp = &svd->vpage[seg_page(seg, seg->s_base + seg->s_size)];
```



```
8509            for (vp = svd->vpage; vp < evp; vp++) {
8510                    VPP_SETPROT(vp, svd->prot);
8511                    VPP_SETADVICE(vp, svd->advice);
8512            }
8513    }
8514 }

8516 /*
8517  * Dump the pages belonging to this segvn segment.
8518  */
8519 static void
8520 segvn_dump(struct seg *seg)
8521 {
8522    struct segvn_data *svd;
8523    page_t *pp;
8524    struct anon_map *amp;
8525    ulong_t anon_index;
8526    struct vnode *vp;
8527    u_offset_t off, offset;
8528    pfn_t pfn;
8529    pgcnt_t page, npages;
8530    caddr_t addr;

8532    npages = seg_pages(seg);
8533    svd = (struct segvn_data *)seg->s_data;
8534    vp = svd->vp;
8535    off = offset = svd->offset;
8536    addr = seg->s_base;

8538    if ((amp = svd->amp) != NULL) {
8539            anon_index = svd->anon_index;
8540            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
8541    }

8543    for (page = 0; page < npages; page++, offset += PAGESIZE) {
8544            struct anon *ap;
8545            int we_own_it = 0;

8547            if (amp && (ap = anon_get_ptr(svd->amp->ahp, anon_index++))) {
8548                    swap_xlate_nopanic(ap, &vp, &off);
8549            } else {
8550                    vp = svd->vp;
8551                    off = offset;
8552            }

8554            /*
8555             * If pp == NULL, the page either does not exist
8556             * or is exclusively locked.  So determine if it
8557             * exists before searching for it.
8558             */

8560            if ((pp = page_lookup_nowait(vp, off, SE_SHARED)))
8561                    we_own_it = 1;
8562            else
8563                    pp = page_exists(vp, off);

8565            if (pp) {
8566                    pfn = page_pptonum(pp);
8567                    dump_addpage(seg->s_as, addr, pfn);
8568                    if (we_own_it)
8569                            page_unlock(pp);
8570            }
8571            addr += PAGESIZE;
8572            dump_timeleft = dump_timeout;
8573    }
```

```
8575         if (amp != NULL)
8576                 ANON_LOCK_EXIT(&amp->a_rwlock);
8577 }

8579 #ifdef DEBUG
8580 static uint32_t segvn_pglock_mtbf = 0;
8581 #endif

8583 #define PCACHE_SHWLIST          ((page_t *)-2)
8584 #define NOPCACHE_SHWLIST        ((page_t *)-1)

8586 /*
8587  * Lock/Unlock anon pages over a given range. Return shadow list. This routine
8588  * uses global segment pcache to cache shadow lists (i.e. pp arrays) of pages
8589  * to avoid the overhead of per page locking, unlocking for subsequent IOs to
8590  * the same parts of the segment. Currently shadow list creation is only
8591  * supported for pure anon segments. MAP_PRIVATE segment pcache entries are
8592  * tagged with segment pointer, starting virtual address and length. This
8593  * approach for MAP_SHARED segments may add many pcache entries for the same
8594  * set of pages and lead to long hash chains that decrease pcache lookup
8595  * performance. To avoid this issue for shared segments shared anon map and
8596  * starting anon index are used for pcache entry tagging. This allows all
8597  * segments to share pcache entries for the same anon range and reduces pcache
8598  * chain's length as well as memory overhead from duplicate shadow lists and
8599  * pcache entries.
8600  *
8601  * softlockcnt field in segvn_data structure counts the number of F_SOFTLOCK'd
8602  * pages via segvn_fault() and pagelock'd pages via this routine. But pagelock
8603  * part of softlockcnt accounting is done differently for private and shared
8604  * segments. In private segment case softlock is only incremented when a new
8605  * shadow list is created but not when an existing one is found via
8606  * seg_plookup(). pcache entries have reference count incremented/decremented
8607  * by each seg_plookup()/seg_pinactive() operation. Only entries that have 0
8608  * reference count can be purged (and purging is needed before segment can be
8609  * freed). When a private segment pcache entry is purged segvn_reclaim() will
8610  * decrement softlockcnt. Since in private segment case each of its pcache
8611  * entries only belongs to this segment we can expect that when
8612  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8613  * segment purge will succeed and softlockcnt will drop to 0. In shared
8614  * segment case reference count in pcache entry counts active locks from many
8615  * different segments so we can't expect segment purging to succeed even when
8616  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8617  * segment. To be able to determine when there're no pending pagelocks in
8618  * shared segment case we don't rely on purging to make softlockcnt drop to 0
8619  * but instead softlockcnt is incremented and decremented for every
8620  * segvn_pagelock(L_PAGELOCK/L_PAGEUNLOCK) call regardless if a new shadow
8621  * list was created or an existing one was found. When softlockcnt drops to 0
8622  * this segment no longer has any claims for pcached shadow lists and the
8623  * segment can be freed even if there're still active pcache entries
8624  * shared by this segment anon map. Shared segment pcache entries belong to
8625  * anon map and are typically removed when anon map is freed after all
8626  * processes destroy the segments that use this anon map.
8627  */
8628 static int
8629 segvn_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***ppp,
8630     enum lock_type type, enum seg_rw rw)
8631 {
8632         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8633         size_t np;
8634         pgcnt_t adjustpages;
8635         pgcnt_t npages;
8636         ulong_t anon_index;
8637         uint_t protchk = (rw == S_READ) ? PROT_READ : PROT_WRITE;
8638         uint_t error;
8639         struct anon_map *amp;
8640         pgcnt_t anpgcnt;
```



```
8641         struct page **pplist, **pl, *pp;
8642         caddr_t a;
8643         size_t page;
8644         caddr_t lpgaddr, lpgeaddr;
8645         anon_sync_obj_t cookie;
8646         int anlock;
8647         struct anon_map *pamp;
8648         caddr_t paddr;
8649         seg_preclaim_cbfunc_t preclaim_callback;
8650         size_t pgsz;
8651         int use_pcache;
8652         size_t wlen;
8653         uint_t pflags = 0;
8654         int sftlck_sbase = 0;
8655         int sftlck_send = 0;

8657 #ifdef DEBUG
8658         if (type == L_PAGELOCK && segvn_pglock_mtbf) {
8659                 hrtime_t ts = gethrtime();
8660                 if ((ts % segvn_pglock_mtbf) == 0) {
8661                         return (ENOTSUP);
8662                 }
8663                 if ((ts % segvn_pglock_mtbf) == 1) {
8664                         return (EFAULT);
8665                 }
8666         }
8667 #endif

8669         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
8670             "segvn_pagelock: start seg %p addr %p", seg, addr);

8672         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
8673         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

8675         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

8677         /*
8678          * for now we only support pagelock to anon memory. We would have to
8679          * check protections for vnode objects and call into the vnode driver.
8680          * That's too much for a fast path. Let the fault entry point handle
8681          * it.
8682          */
8683         if (svd->vp != NULL) {
8684                 if (type == L_PAGELOCK) {
8685                         error = ENOTSUP;
8686                         goto out;
8687                 }
8688                 panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
8689         }
8690         if ((amp = svd->amp) == NULL) {
8691                 if (type == L_PAGELOCK) {
8692                         error = EFAULT;
8693                         goto out;
8694                 }
8695                 panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
8696         }
8697         if (rw != S_READ && rw != S_WRITE) {
8698                 if (type == L_PAGELOCK) {
8699                         error = ENOTSUP;
8700                         goto out;
8701                 }
8702                 panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
8703         }

8705         if (seg->s_szc != 0) {
8706                 /*
```

```
8707                 * We are adjusting the pagelock region to the large page size
8708                 * boundary because the unlocked part of a large page cannot
8709                 * be freed anyway unless all constituent pages of a large
8710                 * page are locked. Bigger regions reduce pcache chain length
8711                 * and improve lookup performance. The tradeoff is that the
8712                 * very first segvn_pagelock() call for a given page is more
8713                 * expensive if only 1 page_t is needed for IO. This is only
8714                 * an issue if pcache entry doesn't get reused by several
8715                 * subsequent calls. We optimize here for the case when pcache
8716                 * is heavily used by repeated IOs to the same address range.
8717                 *
8718                 * Note segment's page size cannot change while we are holding
8719                 * as lock.  And then it cannot change while softlockcnt is
8720                 * not 0. This will allow us to correctly recalculate large
8721                 * page size region for the matching pageunlock/reclaim call
8722                 * since as_pageunlock() caller must always match
8723                 * as_pagelock() call's addr and len.
8724                 *
8725                 * For pageunlock *ppp points to the pointer of page_t that
8726                 * corresponds to the real unadjusted start address. Similar
8727                 * for pagelock *ppp must point to the pointer of page_t that
8728                 * corresponds to the real unadjusted start address.
8729                 */
8730                pgsz = page_get_pagesize(seg->s_szc);
8731                CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
8732                adjustpages = btop((uintptr_t)(addr - lpgaddr));
8733        } else if (len < segvn_pglock_comb_thrshld) {
8734                lpgaddr = addr;
8735                lpgeaddr = addr + len;
8736                adjustpages = 0;
8737                pgsz = PAGESIZE;
8738        } else {
8739                /*
8740                 * Align the address range of large enough requests to allow
8741                 * combining of different shadow lists into 1 to reduce memory
8742                 * overhead from potentially overlapping large shadow lists
8743                 * (worst case is we have a 1MB IO into buffers with start
8744                 * addresses separated by 4K).  Alignment is only possible if
8745                 * padded chunks have sufficient access permissions. Note
8746                 * permissions won't change between L_PAGELOCK and
8747                 * L_PAGEUNLOCK calls since non 0 softlockcnt will force
8748                 * segvn_setprot() to wait until softlockcnt drops to 0. This
8749                 * allows us to determine in L_PAGEUNLOCK the same range we
8750                 * computed in L_PAGELOCK.
8751                 *
8752                 * If alignment is limited by segment ends set
8753                 * sftlck_sbase/sftlck_send flags. In L_PAGELOCK case when
8754                 * these flags are set bump softlockcnt_sbase/softlockcnt_send
8755                 * per segment counters. In L_PAGEUNLOCK case decrease
8756                 * softlockcnt_sbase/softlockcnt_send counters if
8757                 * sftlck_sbase/sftlck_send flags are set.  When
8758                 * softlockcnt_sbase/softlockcnt_send are non 0
8759                 * segvn_concat()/segvn_extend_prev()/segvn_extend_next()
8760                 * won't merge the segments. This restriction combined with
8761                 * restriction on segment unmapping and splitting for segments
8762                 * that have non 0 softlockcnt allows L_PAGEUNLOCK to
8763                 * correctly determine the same range that was previously
8764                 * locked by matching L_PAGELOCK.
8765                 */
8766                pflags = SEGP_PSHIFT | (segvn_pglock_comb_bshift << 16);
8767                pgsz = PAGESIZE;
8768                if (svd->type == MAP_PRIVATE) {
8769                        lpgaddr = (caddr_t)P2ALIGN((uintptr_t)addr,
8770                            segvn_pglock_comb_balign);
8771                        if (lpgaddr < seg->s_base) {
8772                                lpgaddr = seg->s_base;
```



```
8773                                sftlck_sbase = 1;
8774                        }
8775                } else {
8776                        ulong_t aix = svd->anon_index + seg_page(seg, addr);
8777                        ulong_t aaix = P2ALIGN(aix, segvn_pglock_comb_palign);
8778                        if (aaix < svd->anon_index) {
8779                                lpgaddr = seg->s_base;
8780                                sftlck_sbase = 1;
8781                        } else {
8782                                lpgaddr = addr - ptob(aix - aaix);
8783                                ASSERT(lpgaddr >= seg->s_base);
8784                        }
8785                }
8786                if (svd->pageprot && lpgaddr != addr) {
8787                        struct vpage *vp = &svd->vpage[seg_page(seg, lpgaddr)];
8788                        struct vpage *evp = &svd->vpage[seg_page(seg, addr)];
8789                        while (vp < evp) {
8790                                if ((VPP_PROT(vp) & protchk) == 0) {
8791                                        break;
8792                                }
8793                                vp++;
8794                        }
8795                        if (vp < evp) {
8796                                lpgaddr = addr;
8797                                pflags = 0;
8798                        }
8799                }
8800                lpgeaddr = addr + len;
8801                if (pflags) {
8802                        if (svd->type == MAP_PRIVATE) {
8803                                lpgeaddr = (caddr_t)P2ROUNDUP(
8804                                    (uintptr_t)lpgeaddr,
8805                                    segvn_pglock_comb_balign);
8806                        } else {
8807                                ulong_t aix = svd->anon_index +
8808                                    seg_page(seg, lpgeaddr);
8809                                ulong_t aaix = P2ROUNDUP(aix,
8810                                    segvn_pglock_comb_palign);
8811                                if (aaix < aix) {
8812                                        lpgeaddr = 0;
8813                                } else {
8814                                        lpgeaddr += ptob(aaix - aix);
8815                                }
8816                        }
8817                        if (lpgeaddr == 0 ||
8818                            lpgeaddr > seg->s_base + seg->s_size) {
8819                                lpgeaddr = seg->s_base + seg->s_size;
8820                                sftlck_send = 1;
8821                        }
8822                }
8823                if (svd->pageprot && lpgeaddr != addr + len) {
8824                        struct vpage *vp;
8825                        struct vpage *evp;

8827                        vp = &svd->vpage[seg_page(seg, addr + len)];
8828                        evp = &svd->vpage[seg_page(seg, lpgeaddr)];

8830                        while (vp < evp) {
8831                                if ((VPP_PROT(vp) & protchk) == 0) {
8832                                        break;
8833                                }
8834                                vp++;
8835                        }
8836                        if (vp < evp) {
8837                                lpgeaddr = addr + len;
8838                        }
```

```
8839                }
8840                adjustpages = btop((uintptr_t)(addr - lpgaddr));
8841        }

8843        /*
8844         * For MAP_SHARED segments we create pcache entries tagged by amp and
8845         * anon index so that we can share pcache entries with other segments
8846         * that map this amp.  For private segments pcache entries are tagged
8847         * with segment and virtual address.
8848         */
8849        if (svd->type == MAP_SHARED) {
8850                pamp = amp;
8851                paddr = (caddr_t)((lpgaddr - seg->s_base) +
8852                    ptob(svd->anon_index));
8853                preclaim_callback = shamp_reclaim;
8854        } else {
8855                pamp = NULL;
8856                paddr = lpgaddr;
8857                preclaim_callback = segvn_reclaim;
8858        }

8860        if (type == L_PAGEUNLOCK) {
8861                VM_STAT_ADD(segvnvmstats.pagelock[0]);

8863                /*
8864                 * update hat ref bits for /proc. We need to make sure
8865                 * that threads tracing the ref and mod bits of the
8866                 * address space get the right data.
8867                 * Note: page ref and mod bits are updated at reclaim time
8868                 */
8869                if (seg->s_as->a_vbits) {
8870                        for (a = addr; a < addr + len; a += PAGESIZE) {
8871                                if (rw == S_WRITE) {
8872                                        hat_setstat(seg->s_as, a,
8873                                            PAGESIZE, P_REF | P_MOD);
8874                                } else {
8875                                        hat_setstat(seg->s_as, a,
8876                                            PAGESIZE, P_REF);
8877                                }
8878                        }
8879                }

8881                /*
8882                 * Check the shadow list entry after the last page used in
8883                 * this IO request. If it's NOPCACHE_SHWLIST the shadow list
8884                 * was not inserted into pcache and is not large page
8885                 * adjusted.  In this case call reclaim callback directly and
8886                 * don't adjust the shadow list start and size for large
8887                 * pages.
8888                 */
8889                npages = btop(len);
8890                if ((*ppp)[npages] == NOPCACHE_SHWLIST) {
8891                        void *ptag;
8892                        if (pamp != NULL) {
8893                                ASSERT(svd->type == MAP_SHARED);
8894                                ptag = (void *)pamp;
8895                                paddr = (caddr_t)((addr - seg->s_base) +
8896                                    ptob(svd->anon_index));
8897                        } else {
8898                                ptag = (void *)seg;
8899                                paddr = addr;
8900                        }
8901                        (*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
8902                } else {
8903                        ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
8904                            IS_SWAPFSVP((*ppp)[npages]->p_vnode));
```



```
8905                        len = lpgeaddr - lpgaddr;
8906                        npages = btop(len);
8907                        seg_pinactive(seg, pamp, paddr, len,
8908                            *ppp - adjustpages, rw, pflags, preclaim_callback);
8909                }

8911                if (pamp != NULL) {
8912                        ASSERT(svd->type == MAP_SHARED);
8913                        ASSERT(svd->softlockcnt >= npages);
8914                        atomic_add_long((ulong_t *)&svd->softlockcnt, -npages);
8915                }

8917                if (sftlck_sbase) {
8918                        ASSERT(svd->softlockcnt_sbase > 0);
8919                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_sbase);
8920                }
8921                if (sftlck_send) {
8922                        ASSERT(svd->softlockcnt_send > 0);
8923                        atomic_dec_ulong((ulong_t *)&svd->softlockcnt_send);
8924                }

8926                /*
8927                 * If someone is blocked while unmapping, we purge
8928                 * segment page cache and thus reclaim pplist synchronously
8929                 * without waiting for seg_pasync_thread. This speeds up
8930                 * unmapping in cases where munmap(2) is called, while
8931                 * raw async i/o is still in progress or where a thread
8932                 * exits on data fault in a multithreaded application.
8933                 */
8934                if (AS_ISUNMAPWAIT(seg->s_as)) {
8935                        if (svd->softlockcnt == 0) {
8936                                mutex_enter(&seg->s_as->a_contents);
8937                                if (AS_ISUNMAPWAIT(seg->s_as)) {
8938                                        AS_CLRUNMAPWAIT(seg->s_as);
8939                                        cv_broadcast(&seg->s_as->a_cv);
8940                                }
8941                                mutex_exit(&seg->s_as->a_contents);
8942                        } else if (pamp == NULL) {
8943                                /*
8944                                 * softlockcnt is not 0 and this is a
8945                                 * MAP_PRIVATE segment. Try to purge its
8946                                 * pcache entries to reduce softlockcnt.
8947                                 * If it drops to 0 segvn_reclaim()
8948                                 * will wake up a thread waiting on
8949                                 * unmapwait flag.
8950                                 *
8951                                 * We don't purge MAP_SHARED segments with non
8952                                 * 0 softlockcnt since IO is still in progress
8953                                 * for such segments.
8954                                 */
8955                                ASSERT(svd->type == MAP_PRIVATE);
8956                                segvn_purge(seg);
8957                        }
8958                }
8959                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8960                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_UNLOCK_END,
8961                    "segvn_pagelock: unlock seg %p addr %p", seg, addr);
8962                return (0);
8963        }

8965        /* The L_PAGELOCK case ... */

8967        VM_STAT_ADD(segvnvmstats.pagelock[1]);

8969        /*
8970         * For MAP_SHARED segments we have to check protections before
```

```
8971          * seg_plookup() since pcache entries may be shared by many segments
8972          * with potentially different page protections.
8973          */
8974         if (pamp != NULL) {
8975                 ASSERT(svd->type == MAP_SHARED);
8976                 if (svd->pageprot == 0) {
8977                         if ((svd->prot & protchk) == 0) {
8978                                 error = EACCES;
8979                                 goto out;
8980                         }
8981                 } else {
8982                         /*
8983                          * check page protections
8984                          */
8985                         caddr_t ea;

8987                         if (seg->s_szc) {
8988                                 a = lpgaddr;
8989                                 ea = lpgeaddr;
8990                         } else {
8991                                 a = addr;
8992                                 ea = addr + len;
8993                         }
8994                         for (; a < ea; a += pgsz) {
8995                                 struct vpage *vp;

8997                                 ASSERT(seg->s_szc == 0 ||
8998                                     sameprot(seg, a, pgsz));
8999                                 vp = &svd->vpage[seg_page(seg, a)];
9000                                 if ((VPP_PROT(vp) & protchk) == 0) {
9001                                         error = EACCES;
9002                                         goto out;
9003                                 }
9004                         }
9005                 }
9006         }

9008         /*
9009          * try to find pages in segment page cache
9010          */
9011         pplist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
9012         if (pplist != NULL) {
9013                 if (pamp != NULL) {
9014                         npages = btop((uintptr_t)(lpgeaddr - lpgaddr));
9015                         ASSERT(svd->type == MAP_SHARED);
9016                         atomic_add_long((ulong_t *)&svd->softlockcnt,
9017                             npages);
9018                 }
9019                 if (sftlck_sbase) {
9020                         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9021                 }
9022                 if (sftlck_send) {
9023                         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9024                 }
9025                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9026                 *ppp = pplist + adjustpages;
9027                 TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_HIT_END,
9028                     "segvn_pagelock: cache hit seg %p addr %p", seg, addr);
9029                 return (0);
9030         }

9032         /*
9033          * For MAP_SHARED segments we already verified above that segment
9034          * protections allow this pagelock operation.
9035          */
9036         if (pamp == NULL) {
```



```
9037                 ASSERT(svd->type == MAP_PRIVATE);
9038                 if (svd->pageprot == 0) {
9039                         if ((svd->prot & protchk) == 0) {
9040                                 error = EACCES;
9041                                 goto out;
9042                         }
9043                         if (svd->prot & PROT_WRITE) {
9044                                 wlen = lpgeaddr - lpgaddr;
9045                         } else {
9046                                 wlen = 0;
9047                                 ASSERT(rw == S_READ);
9048                         }
9049                 } else {
9050                         int wcont = 1;
9051                         /*
9052                          * check page protections
9053                          */
9054                         for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
9055                                 struct vpage *vp;

9057                                 ASSERT(seg->s_szc == 0 ||
9058                                     sameprot(seg, a, pgsz));
9059                                 vp = &svd->vpage[seg_page(seg, a)];
9060                                 if ((VPP_PROT(vp) & protchk) == 0) {
9061                                         error = EACCES;
9062                                         goto out;
9063                                 }
9064                                 if (wcont && (VPP_PROT(vp) & PROT_WRITE)) {
9065                                         wlen += pgsz;
9066                                 } else {
9067                                         wcont = 0;
9068                                         ASSERT(rw == S_READ);
9069                                 }
9070                         }
9071                 }
9072                 ASSERT(rw == S_READ || wlen == lpgeaddr - lpgaddr);
9073                 ASSERT(rw == S_WRITE || wlen <= lpgeaddr - lpgaddr);
9074         }

9076         /*
9077          * Only build large page adjusted shadow list if we expect to insert
9078          * it into pcache. For large enough pages it's a big overhead to
9079          * create a shadow list of the entire large page. But this overhead
9080          * should be amortized over repeated pcache hits on subsequent reuse
9081          * of this shadow list (IO into any range within this shadow list will
9082          * find it in pcache since we large page align the request for pcache
9083          * lookups). pcache performance is improved with bigger shadow lists
9084          * as it reduces the time to pcache the entire big segment and reduces
9085          * pcache chain length.
9086          */
9087         if (seg_pinsert_check(seg, pamp, paddr,
9088             lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
9089                 addr = lpgaddr;
9090                 len = lpgeaddr - lpgaddr;
9091                 use_pcache = 1;
9092         } else {
9093                 use_pcache = 0;
9094                 /*
9095                  * Since this entry will not be inserted into the pcache, we
9096                  * will not do any adjustments to the starting address or
9097                  * size of the memory to be locked.
9098                  */
9099                 adjustpages = 0;
9100         }
9101         npages = btop(len);
```

```
9103         pplist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
9104         pl = pplist;
9105         *ppp = pplist + adjustpages;
9106         /*
9107          * If use_pcache is 0 this shadow list is not large page adjusted.
9108          * Record this info in the last entry of shadow array so that
9109          * L_PAGEUNLOCK can determine if it should large page adjust the
9110          * address range to find the real range that was locked.
9111          */
9112         pl[npages] = use_pcache ? PCACHE_SHWLIST : NOPCACHE_SHWLIST;

9114         page = seg_page(seg, addr);
9115         anon_index = svd->anon_index + page;

9117         anlock = 0;
9118         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
9119         ASSERT(amp->a_szc >= seg->s_szc);
9120         anpgcnt = page_get_pagecnt(amp->a_szc);
9121         for (a = addr; a < addr + len; a += PAGESIZE, anon_index++) {
9122                 struct anon *ap;
9123                 struct vnode *vp;
9124                 u_offset_t off;

9126                 /*
9127                  * Lock and unlock anon array only once per large page.
9128                  * anon_array_enter() locks the root anon slot according to
9129                  * a_szc which can't change while anon map is locked.  We lock
9130                  * anon the first time through this loop and each time we
9131                  * reach anon index that corresponds to a root of a large
9132                  * page.
9133                  */
9134                 if (a == addr || P2PHASE(anon_index, anpgcnt) == 0) {
9135                         ASSERT(anlock == 0);
9136                         anon_array_enter(amp, anon_index, &cookie);
9137                         anlock = 1;
9138                 }
9139                 ap = anon_get_ptr(amp->ahp, anon_index);

9141                 /*
9142                  * We must never use seg_pcache for COW pages
9143                  * because we might end up with original page still
9144                  * lying in seg_pcache even after private page is
9145                  * created. This leads to data corruption as
9146                  * aio_write refers to the page still in cache
9147                  * while all other accesses refer to the private
9148                  * page.
9149                  */
9150                 if (ap == NULL || ap->an_refcnt != 1) {
9151                         struct vpage *vpage;

9153                         if (seg->s_szc) {
9154                                 error = EFAULT;
9155                                 break;
9156                         }
9157                         if (svd->vpage != NULL) {
9158                                 vpage = &svd->vpage[seg_page(seg, a)];
9159                         } else {
9160                                 vpage = NULL;
9161                         }
9162                         ASSERT(anlock);
9163                         anon_array_exit(&cookie);
9164                         anlock = 0;
9165                         pp = NULL;
9166                         error = segvn_faultpage(seg->s_as->a_hat, seg, a, 0,
9167                             vpage, &pp, 0, F_INVAL, rw, 1);
9168                         if (error) {
```



```
9169                                 error = fc_decode(error);
9170                                 break;
9171                         }
9172                         anon_array_enter(amp, anon_index, &cookie);
9173                         anlock = 1;
9174                         ap = anon_get_ptr(amp->ahp, anon_index);
9175                         if (ap == NULL || ap->an_refcnt != 1) {
9176                                 error = EFAULT;
9177                                 break;
9178                         }
9179                 }
9180                 swap_xlate(ap, &vp, &off);
9181                 pp = page_lookup_nowait(vp, off, SE_SHARED);
9182                 if (pp == NULL) {
9183                         error = EFAULT;
9184                         break;
9185                 }
9186                 if (ap->an_pvp != NULL) {
9187                         anon_swap_free(ap, pp);
9188                 }
9189                 /*
9190                  * Unlock anon if this is the last slot in a large page.
9191                  */
9192                 if (P2PHASE(anon_index, anpgcnt) == anpgcnt - 1) {
9193                         ASSERT(anlock);
9194                         anon_array_exit(&cookie);
9195                         anlock = 0;
9196                 }
9197                 *pplist++ = pp;
9198         }
9199         if (anlock) {           /* Ensure the lock is dropped */
9200                 anon_array_exit(&cookie);
9201         }
9202         ANON_LOCK_EXIT(&amp->a_rwlock);

9204         if (a >= addr + len) {
9205                 atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
9206                 if (pamp != NULL) {
9207                         ASSERT(svd->type == MAP_SHARED);
9208                         atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
9209                             npages);
9210                         wlen = len;
9211                 }
9212                 if (sftlck_sbase) {
9213                         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9214                 }
9215                 if (sftlck_send) {
9216                         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9217                 }
9218                 if (use_pcache) {
9219                         (void) seg_pinsert(seg, pamp, paddr, len, wlen, pl,
9220                             rw, pflags, preclaim_callback);
9221                 }
9222                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9223                 TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
9224                     "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
9225                 return (0);
9226         }

9228         pplist = pl;
9229         np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
9230         while (np > (uint_t)0) {
9231                 ASSERT(PAGE_LOCKED(*pplist));
9232                 page_unlock(*pplist);
9233                 np--;
9234                 pplist++;
```

```
9235         }
9236         kmem_free(pl, sizeof (page_t *) * (npages + 1));
9237 out:
9238         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9239         *ppp = NULL;
9240         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
9241             "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
9242         return (error);
9243 }

9245 /*
9246  * purge any cached pages in the I/O page cache
9247  */
9248 static void
9249 segvn_purge(struct seg *seg)
9250 {
9251         struct segvn_data *svd = (struct segvn_data *)seg->s_data;

9253         /*
9254          * pcache is only used by pure anon segments.
9255          */
9256         if (svd->amp == NULL || svd->vp != NULL) {
9257                 return;
9258         }

9260         /*
9261          * For MAP_SHARED segments non 0 segment's softlockcnt means
9262          * active IO is still in progress via this segment. So we only
9263          * purge MAP_SHARED segments when their softlockcnt is 0.
9264          */
9265         if (svd->type == MAP_PRIVATE) {
9266                 if (svd->softlockcnt) {
9267                         seg_ppurge(seg, NULL, 0);
9268                 }
9269         } else if (svd->softlockcnt == 0 && svd->amp->a_softlockcnt != 0) {
9270                 seg_ppurge(seg, svd->amp, 0);
9271         }
9272 }

9274 /*
9275  * If async argument is not 0 we are called from pcache async thread and don't
9276  * hold AS lock.
9277  */

9279 /*ARGSUSED*/
9280 static int
9281 segvn_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
9282         enum seg_rw rw, int async)
9283 {
9284         struct seg *seg = (struct seg *)ptag;
9285         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9286         pgcnt_t np, npages;
9287         struct page **pl;

9289         npages = np = btop(len);
9290         ASSERT(npages);

9292         ASSERT(svd->vp == NULL && svd->amp != NULL);
9293         ASSERT(svd->softlockcnt >= npages);
9294         ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

9296         pl = pplist;

9298         ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
9299         ASSERT(!async || pl[np] == PCACHE_SHWLIST);
```



```
9301         while (np > (uint_t)0) {
9302                 if (rw == S_WRITE) {
9303                         hat_setrefmod(*pplist);
9304                 } else {
9305                         hat_setref(*pplist);
9306                 }
9307                 page_unlock(*pplist);
9308                 np--;
9309                 pplist++;
9310         }

9312         kmem_free(pl, sizeof (page_t *) * (npages + 1));

9314         /*
9315          * If we are pcache async thread we don't hold AS lock. This means if
9316          * softlockcnt drops to 0 after the decrement below address space may
9317          * get freed. We can't allow it since after softlock derement to 0 we
9318          * still need to access as structure for possible wakeup of unmap
9319          * waiters. To prevent the disappearance of as we take this segment
9320          * segfree_syncmtx. segvn_free() also takes this mutex as a barrier to
9321          * make sure this routine completes before segment is freed.
9322          *
9323          * The second complication we have to deal with in async case is a
9324          * possibility of missed wake up of unmap wait thread. When we don't
9325          * hold as lock here we may take a_contents lock before unmap wait
9326          * thread that was first to see softlockcnt was still not 0. As a
9327          * result we'll fail to wake up an unmap wait thread. To avoid this
9328          * race we set nounmapwait flag in as structure if we drop softlockcnt
9329          * to 0 when we were called by pcache async thread.  unmapwait thread
9330          * will not block if this flag is set.
9331          */
9332         if (async) {
9333                 mutex_enter(&svd->segfree_syncmtx);
9334         }

9336         if (!atomic_add_long_nv((ulong_t *)&svd->softlockcnt, -npages)) {
9337                 if (async || AS_ISUNMAPWAIT(seg->s_as)) {
9338                         mutex_enter(&seg->s_as->a_contents);
9339                         if (async) {
9340                                 AS_SETNOUNMAPWAIT(seg->s_as);
9341                         }
9342                         if (AS_ISUNMAPWAIT(seg->s_as)) {
9343                                 AS_CLRUNMAPWAIT(seg->s_as);
9344                                 cv_broadcast(&seg->s_as->a_cv);
9345                         }
9346                         mutex_exit(&seg->s_as->a_contents);
9347                 }
9348         }

9350         if (async) {
9351                 mutex_exit(&svd->segfree_syncmtx);
9352         }
9353         return (0);
9354 }

9356 /*ARGSUSED*/
9357 static int
9358 shamp_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
9359         enum seg_rw rw, int async)
9360 {
9361         amp_t *amp = (amp_t *)ptag;
9362         pgcnt_t np, npages;
9363         struct page **pl;

9365         npages = np = btop(len);
9366         ASSERT(npages);
```

```
9367         ASSERT(amp->a_softlockcnt >= npages);

9369         pl = pplist;

9371         ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
9372         ASSERT(!async || pl[np] == PCACHE_SHWLIST);

9374         while (np > (uint_t)0) {
9375                 if (rw == S_WRITE) {
9376                         hat_setrefmod(*pplist);
9377                 } else {
9378                         hat_setref(*pplist);
9379                 }
9380                 page_unlock(*pplist);
9381                 np--;
9382                 pplist++;
9383         }

9385         kmem_free(pl, sizeof (page_t *) * (npages + 1));

9387         /*
9388          * If somebody sleeps in anonmap_purge() wake them up if a_softlockcnt
9389          * drops to 0. anon map can't be freed until a_softlockcnt drops to 0
9390          * and anonmap_purge() acquires a_purgemtx.
9391          */
9392         mutex_enter(&amp->a_purgemtx);
9393         if (!atomic_add_long_nv((ulong_t *)&amp->a_softlockcnt, -npages) &&
9394             amp->a_purgewait) {
9395                 amp->a_purgewait = 0;
9396                 cv_broadcast(&amp->a_purgecv);
9397         }
9398         mutex_exit(&amp->a_purgemtx);
9399         return (0);
9400 }

9402 /*
9403  * get a memory ID for an addr in a given segment
9404  *
9405  * XXX only creates PAGESIZE pages if anon slots are not initialized.
9406  * At fault time they will be relocated into larger pages.
9407  */
9408 static int
9409 segvn_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
9410 {
9411         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9412         struct anon     *ap = NULL;
9413         ulong_t         anon_index;
9414         struct anon_map *amp;
9415         anon_sync_obj_t cookie;

9417         if (svd->type == MAP_PRIVATE) {
9418                 memidp->val[0] = (uintptr_t)seg->s_as;
9419                 memidp->val[1] = (uintptr_t)addr;
9420                 return (0);
9421         }

9423         if (svd->type == MAP_SHARED) {
9424                 if (svd->vp) {
9425                         memidp->val[0] = (uintptr_t)svd->vp;
9426                         memidp->val[1] = (u_longlong_t)svd->offset +
9427                             (uintptr_t)(addr - seg->s_base);
9428                         return (0);
9429                 } else {

9431                         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
9432                         if ((amp = svd->amp) != NULL) {
```



```
9433                                 anon_index = svd->anon_index +
9434                                     seg_page(seg, addr);
9435                         }
9436                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

9438                         ASSERT(amp != NULL);

9440                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
9441                         anon_array_enter(amp, anon_index, &cookie);
9442                         ap = anon_get_ptr(amp->ahp, anon_index);
9443                         if (ap == NULL) {
9444                                 page_t          *pp;

9446                                 pp = anon_zero(seg, addr, &ap, svd->cred);
9447                                 if (pp == NULL) {
9448                                         anon_array_exit(&cookie);
9449                                         ANON_LOCK_EXIT(&amp->a_rwlock);
9450                                         return (ENOMEM);
9451                                 }
9452                                 ASSERT(anon_get_ptr(amp->ahp, anon_index)
9453                                     == NULL);
9454                                 (void) anon_set_ptr(amp->ahp, anon_index,
9455                                     ap, ANON_SLEEP);
9456                                 page_unlock(pp);
9457                         }

9459                         anon_array_exit(&cookie);
9460                         ANON_LOCK_EXIT(&amp->a_rwlock);

9462                         memidp->val[0] = (uintptr_t)ap;
9463                         memidp->val[1] = (uintptr_t)addr & PAGEOFFSET;
9464                         return (0);
9465                 }
9466         }
9467         return (EINVAL);
9468 }

9470 static int
9471 sameprot(struct seg *seg, caddr_t a, size_t len)
9472 {
9473         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9474         struct vpage *vpage;
9475         spgcnt_t pages = btop(len);
9476         uint_t prot;

9478         if (svd->pageprot == 0)
9479                 return (1);

9481         ASSERT(svd->vpage != NULL);

9483         vpage = &svd->vpage[seg_page(seg, a)];
9484         prot = VPP_PROT(vpage);
9485         vpage++;
9486         pages--;
9487         while (pages-- > 0) {
9488                 if (prot != VPP_PROT(vpage))
9489                         return (0);
9490                 vpage++;
9491         }
9492         return (1);
9493 }

9495 /*
9496  * Get memory allocation policy info for specified address in given segment
9497  */
9498 static lgrp_mem_policy_info_t *
```

```
9499 segvn_getpolicy(struct seg *seg, caddr_t addr)
9500 {
9501         struct anon_map         *amp;
9502         ulong_t                 anon_index;
9503         lgrp_mem_policy_info_t  *policy_info;
9504         struct segvn_data       *svn_data;
9505         u_offset_t              vn_off;
9506         vnode_t                 *vp;

9508         ASSERT(seg != NULL);

9510         svn_data = (struct segvn_data *)seg->s_data;
9511         if (svn_data == NULL)
9512                 return (NULL);

9514         /*
9515          * Get policy info for private or shared memory
9516          */
9517         if (svn_data->type != MAP_SHARED) {
9518                 if (svn_data->tr_state != SEGVN_TR_ON) {
9519                         policy_info = &svn_data->policy_info;
9520                 } else {
9521                         policy_info = &svn_data->tr_policy_info;
9522                         ASSERT(policy_info->mem_policy ==
9523                             LGRP_MEM_POLICY_NEXT_SEG);
9524                 }
9525         } else {
9526                 amp = svn_data->amp;
9527                 anon_index = svn_data->anon_index + seg_page(seg, addr);
9528                 vp = svn_data->vp;
9529                 vn_off = svn_data->offset + (uintptr_t)(addr - seg->s_base);
9530                 policy_info = lgrp_shm_policy_get(amp, anon_index, vp, vn_off);
9531         }

9533         return (policy_info);
9534 }

9536 /*ARGSUSED*/
9537 static int
9538 segvn_capable(struct seg *seg, segcapability_t capability)
9539 {
9540         return (0);
9541 }

9543 /*
9544  * Bind text vnode segment to an amp. If we bind successfully mappings will be
9545  * established to per vnode mapping per lgroup amp pages instead of to vnode
9546  * pages. There's one amp per vnode text mapping per lgroup. Many processes
9547  * may share the same text replication amp. If a suitable amp doesn't already
9548  * exist in svntr hash table create a new one.  We may fail to bind to amp if
9549  * segment is not eligible for text replication.  Code below first checks for
9550  * these conditions. If binding is successful segment tr_state is set to on
9551  * and svd->amp points to the amp to use. Otherwise tr_state is set to off and
9552  * svd->amp remains as NULL.
9553  */
9554 static void
9555 segvn_textrepl(struct seg *seg)
9556 {
9557         struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
9558         vnode_t                 *vp = svd->vp;
9559         u_offset_t              off = svd->offset;
9560         size_t                  size = seg->s_size;
9561         u_offset_t              eoff = off + size;
9562         uint_t                  szc = seg->s_szc;
9563         ulong_t                 hash = SVNTR_HASH_FUNC(vp);
9564         svntr_t                 *svntrp;
```



```
9565         struct vattr            va;
9566         proc_t                  *p = seg->s_as->a_proc;
9567         lgrp_id_t               lgrp_id;
9568         lgrp_id_t               olid;
9569         int                     first;
9570         struct anon_map         *amp;

9572         ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9573         ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9574         ASSERT(p != NULL);
9575         ASSERT(svd->tr_state == SEGVN_TR_INIT);
9576         ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9577         ASSERT(svd->flags & MAP_TEXT);
9578         ASSERT(svd->type == MAP_PRIVATE);
9579         ASSERT(vp != NULL && svd->amp == NULL);
9580         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
9581         ASSERT(!(svd->flags & MAP_NORESERVE) && svd->swresv == 0);
9582         ASSERT(seg->s_as != &kas);
9583         ASSERT(off < eoff);
9584         ASSERT(svntr_hashtab != NULL);

9586         /*
9587          * If numa optimizations are no longer desired bail out.
9588          */
9589         if (!lgrp_optimizations()) {
9590                 svd->tr_state = SEGVN_TR_OFF;
9591                 return;
9592         }

9594         /*
9595          * Avoid creating anon maps with size bigger than the file size.
9596          * If VOP_GETATTR() call fails bail out.
9597          */
9598         va.va_mask = AT_SIZE | AT_MTIME | AT_CTIME;
9599         if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL) != 0) {
9600                 svd->tr_state = SEGVN_TR_OFF;
9601                 SEGVN_TR_ADDSTAT(gaerr);
9602                 return;
9603         }
9604         if (btopr(va.va_size) < btopr(eoff)) {
9605                 svd->tr_state = SEGVN_TR_OFF;
9606                 SEGVN_TR_ADDSTAT(overmap);
9607                 return;
9608         }

9610         /*
9611          * VVMEXEC may not be set yet if exec() prefaults text segment. Set
9612          * this flag now before vn_is_mapped(V_WRITE) so that MAP_SHARED
9613          * mapping that checks if trcache for this vnode needs to be
9614          * invalidated can't miss us.
9615          */
9616         if (!(vp->v_flag & VVMEXEC)) {
9617                 mutex_enter(&vp->v_lock);
9618                 vp->v_flag |= VVMEXEC;
9619                 mutex_exit(&vp->v_lock);
9620         }
9621         mutex_enter(&svntr_hashtab[hash].tr_lock);
9622         /*
9623          * Bail out if potentially MAP_SHARED writable mappings exist to this
9624          * vnode.  We don't want to use old file contents from existing
9625          * replicas if this mapping was established after the original file
9626          * was changed.
9627          */
9628         if (vn_is_mapped(vp, V_WRITE)) {
9629                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9630                 svd->tr_state = SEGVN_TR_OFF;
```

```
9631                    SEGVN_TR_ADDSTAT(wrcnt);
9632                    return;
9633            }
9634            svntrp = svntr_hashtab[hash].tr_head;
9635            for (; svntrp != NULL; svntrp = svntrp->tr_next) {
9636                    ASSERT(svntrp->tr_refcnt != 0);
9637                    if (svntrp->tr_vp != vp) {
9638                            continue;
9639                    }
9641                    /*
9642                     * Bail out if the file or its attributes were changed after
9643                     * this replication entry was created since we need to use the
9644                     * latest file contents. Note that mtime test alone is not
9645                     * sufficient because a user can explicitly change mtime via
9646                     * utimes(2) interfaces back to the old value after modifiying
9647                     * the file contents. To detect this case we also have to test
9648                     * ctime which among other things records the time of the last
9649                     * mtime change by utimes(2). ctime is not changed when the file
9650                     * is only read or executed so we expect that typically existing
9651                     * replication amp's can be used most of the time.
9652                     */
9653                    if (!svntrp->tr_valid ||
9654                        svntrp->tr_mtime.tv_sec != va.va_mtime.tv_sec ||
9655                        svntrp->tr_mtime.tv_nsec != va.va_mtime.tv_nsec ||
9656                        svntrp->tr_ctime.tv_sec != va.va_ctime.tv_sec ||
9657                        svntrp->tr_ctime.tv_nsec != va.va_ctime.tv_nsec) {
9658                            mutex_exit(&svntr_hashtab[hash].tr_lock);
9659                            svd->tr_state = SEGVN_TR_OFF;
9660                            SEGVN_TR_ADDSTAT(stale);
9661                            return;
9662                    }
9663                    /*
9664                     * if off, eoff and szc match current segment we found the
9665                     * existing entry we can use.
9666                     */
9667                    if (svntrp->tr_off == off && svntrp->tr_eoff == eoff &&
9668                        svntrp->tr_szc == szc) {
9669                            break;
9670                    }
9671                    /*
9672                     * Don't create different but overlapping in file offsets
9673                     * entries to avoid replication of the same file pages more
9674                     * than once per lgroup.
9675                     */
9676                    if ((off >= svntrp->tr_off && off < svntrp->tr_eoff) ||
9677                        (eoff > svntrp->tr_off && eoff <= svntrp->tr_eoff)) {
9678                            mutex_exit(&svntr_hashtab[hash].tr_lock);
9679                            svd->tr_state = SEGVN_TR_OFF;
9680                            SEGVN_TR_ADDSTAT(overlap);
9681                            return;
9682                    }
9683            }
9684            /*
9685             * If we didn't find existing entry create a new one.
9686             */
9687            if (svntrp == NULL) {
9688                    svntrp = kmem_cache_alloc(svntr_cache, KM_NOSLEEP);
9689                    if (svntrp == NULL) {
9690                            mutex_exit(&svntr_hashtab[hash].tr_lock);
9691                            svd->tr_state = SEGVN_TR_OFF;
9692                            SEGVN_TR_ADDSTAT(nokmem);
9693                            return;
9694                    }
9695 #ifdef DEBUG
9696                    {
```



```
9697                            lgrp_id_t i;
9698                            for (i = 0; i < NLGRPS_MAX; i++) {
9699                                    ASSERT(svntrp->tr_amp[i] == NULL);
9700                            }
9701                    }
9702 #endif /* DEBUG */
9703                    svntrp->tr_vp = vp;
9704                    svntrp->tr_off = off;
9705                    svntrp->tr_eoff = eoff;
9706                    svntrp->tr_szc = szc;
9707                    svntrp->tr_valid = 1;
9708                    svntrp->tr_mtime = va.va_mtime;
9709                    svntrp->tr_ctime = va.va_ctime;
9710                    svntrp->tr_refcnt = 0;
9711                    svntrp->tr_next = svntr_hashtab[hash].tr_head;
9712                    svntr_hashtab[hash].tr_head = svntrp;
9713            }
9714            first = 1;
9715 again:
9716            /*
9717             * We want to pick a replica with pages on main thread's (t_tid = 1,
9718             * aka T1) lgrp. Currently text replication is only optimized for
9719             * workloads that either have all threads of a process on the same
9720             * lgrp or execute their large text primarily on main thread.
9721             */
9722            lgrp_id = p->p_t1_lgrpid;
9723            if (lgrp_id == LGRP_NONE) {
9724                    /*
9725                     * In case exec() prefaults text on non main thread use
9726                     * current thread lgrpid.  It will become main thread anyway
9727                     * soon.
9728                     */
9729                    lgrp_id = lgrp_home_id(curthread);
9730            }
9731            /*
9732             * Set p_tr_lgrpid to lgrpid if it hasn't been set yet.  Otherwise
9733             * just set it to NLGRPS_MAX if it's different from current process T1
9734             * home lgrp.  p_tr_lgrpid is used to detect if process uses text
9735             * replication and T1 new home is different from lgrp used for text
9736             * replication. When this happens asyncronous segvn thread rechecks if
9737             * segments should change lgrps used for text replication.  If we fail
9738             * to set p_tr_lgrpid with atomic_cas_32 then set it to NLGRPS_MAX
9739             * without cas if it's not already NLGRPS_MAX and not equal lgrp_id
9740             * we want to use.  We don't need to use cas in this case because
9741             * another thread that races in between our non atomic check and set
9742             * may only change p_tr_lgrpid to NLGRPS_MAX at this point.
9743             */
9744            ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9745            olid = p->p_tr_lgrpid;
9746            if (lgrp_id != olid && olid != NLGRPS_MAX) {
9747                    lgrp_id_t nlid = (olid == LGRP_NONE) ? lgrp_id : NLGRPS_MAX;
9748                    if (atomic_cas_32((uint32_t *)&p->p_tr_lgrpid, olid, nlid) !=
9749                        olid) {
9750                            olid = p->p_tr_lgrpid;
9751                            ASSERT(olid != LGRP_NONE);
9752                            if (olid != lgrp_id && olid != NLGRPS_MAX) {
9753                                    p->p_tr_lgrpid = NLGRPS_MAX;
9754                            }
9755                    }
9756                    ASSERT(p->p_tr_lgrpid != LGRP_NONE);
9757                    membar_producer();
9758                    /*
9759                     * lgrp_move_thread() won't schedule async recheck after
9760                     * p->p_t1_lgrpid update unless p->p_tr_lgrpid is not
9761                     * LGRP_NONE. Recheck p_t1_lgrpid once now that p->p_tr_lgrpid
9762                     * is not LGRP_NONE.
```

```
9763                 */
9764                if (first && p->p_t1_lgrpid != LGRP_NONE &&
9765                    p->p_t1_lgrpid != lgrp_id) {
9766                        first = 0;
9767                        goto again;
9768                }
9769        }
9770        /*
9771         * If no amp was created yet for lgrp_id create a new one as long as
9772         * we have enough memory to afford it.
9773         */
9774        if ((amp = svntrp->tr_amp[lgrp_id]) == NULL) {
9775                size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
9776                if (trmem > segvn_textrepl_max_bytes) {
9777                        SEGVN_TR_ADDSTAT(normem);
9778                        goto fail;
9779                }
9780                if (anon_try_resv_zone(size, NULL) == 0) {
9781                        SEGVN_TR_ADDSTAT(noanon);
9782                        goto fail;
9783                }
9784                amp = anonmap_alloc(size, size, ANON_NOSLEEP);
9785                if (amp == NULL) {
9786                        anon_unresv_zone(size, NULL);
9787                        SEGVN_TR_ADDSTAT(nokmem);
9788                        goto fail;
9789                }
9790                ASSERT(amp->refcnt == 1);
9791                amp->a_szc = szc;
9792                svntrp->tr_amp[lgrp_id] = amp;
9793                SEGVN_TR_ADDSTAT(newamp);
9794        }
9795        svntrp->tr_refcnt++;
9796        ASSERT(svd->svn_trnext == NULL);
9797        ASSERT(svd->svn_trprev == NULL);
9798        svd->svn_trnext = svntrp->tr_svnhead;
9799        svd->svn_trprev = NULL;
9800        if (svntrp->tr_svnhead != NULL) {
9801                svntrp->tr_svnhead->svn_trprev = svd;
9802        }
9803        svntrp->tr_svnhead = svd;
9804        ASSERT(amp->a_szc == szc && amp->size == size && amp->swresv == size);
9805        ASSERT(amp->refcnt >= 1);
9806        svd->amp = amp;
9807        svd->anon_index = 0;
9808        svd->tr_policy_info.mem_policy = LGRP_MEM_POLICY_NEXT_SEG;
9809        svd->tr_policy_info.mem_lgrpid = lgrp_id;
9810        svd->tr_state = SEGVN_TR_ON;
9811        mutex_exit(&svntr_hashtab[hash].tr_lock);
9812        SEGVN_TR_ADDSTAT(repl);
9813        return;
9814 fail:
9815        ASSERT(segvn_textrepl_bytes >= size);
9816        atomic_add_long(&segvn_textrepl_bytes, -size);
9817        ASSERT(svntrp != NULL);
9818        ASSERT(svntrp->tr_amp[lgrp_id] == NULL);
9819        if (svntrp->tr_refcnt == 0) {
9820                ASSERT(svntrp == svntr_hashtab[hash].tr_head);
9821                svntr_hashtab[hash].tr_head = svntrp->tr_next;
9822                mutex_exit(&svntr_hashtab[hash].tr_lock);
9823                kmem_cache_free(svntr_cache, svntrp);
9824        } else {
9825                mutex_exit(&svntr_hashtab[hash].tr_lock);
9826        }
9827        svd->tr_state = SEGVN_TR_OFF;
9828 }
```



```
9830 /*
9831  * Convert seg back to regular vnode mapping seg by unbinding it from its text
9832  * replication amp.  This routine is most typically called when segment is
9833  * unmapped but can also be called when segment no longer qualifies for text
9834  * replication (e.g. due to protection changes). If unload_unmap is set use
9835  * HAT_UNLOAD_UNMAP flag in hat_unload_callback().  If we are the last user of
9836  * svntr free all its anon maps and remove it from the hash table.
9837  */
9838 static void
9839 segvn_textunrepl(struct seg *seg, int unload_unmap)
9840 {
9841        struct segvn_data       *svd = (struct segvn_data *)seg->s_data;
9842        vnode_t                 *vp = svd->vp;
9843        u_offset_t              off = svd->offset;
9844        size_t                  size = seg->s_size;
9845        u_offset_t              eoff = off + size;
9846        uint_t                  szc = seg->s_szc;
9847        ulong_t                 hash = SVNTR_HASH_FUNC(vp);
9848        svntr_t                 *svntrp;
9849        svntr_t                 **prv_svntrp;
9850        lgrp_id_t               lgrp_id = svd->tr_policy_info.mem_lgrpid;
9851        lgrp_id_t               i;

9853        ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9854        ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
9855            SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9856        ASSERT(svd->tr_state == SEGVN_TR_ON);
9857        ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9858        ASSERT(svd->amp != NULL);
9859        ASSERT(svd->amp->refcnt >= 1);
9860        ASSERT(svd->anon_index == 0);
9861        ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9862        ASSERT(svntr_hashtab != NULL);

9864        mutex_enter(&svntr_hashtab[hash].tr_lock);
9865        prv_svntrp = &svntr_hashtab[hash].tr_head;
9866        for (; (svntrp = *prv_svntrp) != NULL; prv_svntrp = &svntrp->tr_next) {
9867                ASSERT(svntrp->tr_refcnt != 0);
9868                if (svntrp->tr_vp == vp && svntrp->tr_off == off &&
9869                    svntrp->tr_eoff == eoff && svntrp->tr_szc == szc) {
9870                        break;
9871                }
9872        }
9873        if (svntrp == NULL) {
9874                panic("segvn_textunrepl: svntr record not found");
9875        }
9876        if (svntrp->tr_amp[lgrp_id] != svd->amp) {
9877                panic("segvn_textunrepl: amp mismatch");
9878        }
9879        svd->tr_state = SEGVN_TR_OFF;
9880        svd->amp = NULL;
9881        if (svd->svn_trprev == NULL) {
9882                ASSERT(svntrp->tr_svnhead == svd);
9883                svntrp->tr_svnhead = svd->svn_trnext;
9884                if (svntrp->tr_svnhead != NULL) {
9885                        svntrp->tr_svnhead->svn_trprev = NULL;
9886                }
9887                svd->svn_trnext = NULL;
9888        } else {
9889                svd->svn_trprev->svn_trnext = svd->svn_trnext;
9890                if (svd->svn_trnext != NULL) {
9891                        svd->svn_trnext->svn_trprev = svd->svn_trprev;
9892                        svd->svn_trnext = NULL;
9893                }
9894                svd->svn_trprev = NULL;
```

```
9895         }
9896         if (--svntrp->tr_refcnt) {
9897                 mutex_exit(&svntr_hashtab[hash].tr_lock);
9898                 goto done;
9899         }
9900         *prv_svntrp = svntrp->tr_next;
9901         mutex_exit(&svntr_hashtab[hash].tr_lock);
9902         for (i = 0; i < NLGRPS_MAX; i++) {
9903                 struct anon_map *amp = svntrp->tr_amp[i];
9904                 if (amp == NULL) {
9905                         continue;
9906                 }
9907                 ASSERT(amp->refcnt == 1);
9908                 ASSERT(amp->swresv == size);
9909                 ASSERT(amp->size == size);
9910                 ASSERT(amp->a_szc == szc);
9911                 if (amp->a_szc != 0) {
9912                         anon_free_pages(amp->ahp, 0, size, szc);
9913                 } else {
9914                         anon_free(amp->ahp, 0, size);
9915                 }
9916                 svntrp->tr_amp[i] = NULL;
9917                 ASSERT(segvn_textrepl_bytes >= size);
9918                 atomic_add_long(&segvn_textrepl_bytes, -size);
9919                 anon_unresv_zone(amp->swresv, NULL);
9920                 amp->refcnt = 0;
9921                 anonmap_free(amp);
9922         }
9923         kmem_cache_free(svntr_cache, svntrp);
9924 done:
9925         hat_unload_callback(seg->s_as->a_hat, seg->s_base, size,
9926             unload_unmap ? HAT_UNLOAD_UNMAP : 0, NULL);
9927 }

9929 /*
9930  * This is called when a MAP_SHARED writable mapping is created to a vnode
9931  * that is currently used for execution (VVMEXEC flag is set). In this case we
9932  * need to prevent further use of existing replicas.
9933  */
9934 static void
9935 segvn_inval_trcache(vnode_t *vp)
9936 {
9937         ulong_t                 hash = SVNTR_HASH_FUNC(vp);
9938         svntr_t                 *svntrp;

9940         ASSERT(vp->v_flag & VVMEXEC);

9942         if (svntr_hashtab == NULL) {
9943                 return;
9944         }

9946         mutex_enter(&svntr_hashtab[hash].tr_lock);
9947         svntrp = svntr_hashtab[hash].tr_head;
9948         for (; svntrp != NULL; svntrp = svntrp->tr_next) {
9949                 ASSERT(svntrp->tr_refcnt != 0);
9950                 if (svntrp->tr_vp == vp && svntrp->tr_valid) {
9951                         svntrp->tr_valid = 0;
9952                 }
9953         }
9954         mutex_exit(&svntr_hashtab[hash].tr_lock);
9955 }

9957 static void
9958 segvn_trasync_thread(void)
9959 {
9960         callb_cpr_t cpr_info;
```



```
9961         kmutex_t cpr_lock;      /* just for CPR stuff */

9963         mutex_init(&cpr_lock, NULL, MUTEX_DEFAULT, NULL);

9965         CALLB_CPR_INIT(&cpr_info, &cpr_lock,
9966             callb_generic_cpr, "segvn_async");

9968         if (segvn_update_textrepl_interval == 0) {
9969                 segvn_update_textrepl_interval = segvn_update_tr_time * hz;
9970         } else {
9971                 segvn_update_textrepl_interval *= hz;
9972         }
9973         (void) timeout(segvn_trupdate_wakeup, NULL,
9974             segvn_update_textrepl_interval);

9976         for (;;) {
9977                 mutex_enter(&cpr_lock);
9978                 CALLB_CPR_SAFE_BEGIN(&cpr_info);
9979                 mutex_exit(&cpr_lock);
9980                 sema_p(&segvn_trasync_sem);
9981                 mutex_enter(&cpr_lock);
9982                 CALLB_CPR_SAFE_END(&cpr_info, &cpr_lock);
9983                 mutex_exit(&cpr_lock);
9984                 segvn_trupdate();
9985         }
9986 }

9988 static uint64_t segvn_lgrp_trthr_migrs_snpsht = 0;

9990 static void
9991 segvn_trupdate_wakeup(void *dummy)
9992 {
9993         uint64_t cur_lgrp_trthr_migrs = lgrp_get_trthr_migrations();

9995         if (cur_lgrp_trthr_migrs != segvn_lgrp_trthr_migrs_snpsht) {
9996                 segvn_lgrp_trthr_migrs_snpsht = cur_lgrp_trthr_migrs;
9997                 sema_v(&segvn_trasync_sem);
9998         }

10000         if (!segvn_disable_textrepl_update &&
10001             segvn_update_textrepl_interval != 0) {
10002                 (void) timeout(segvn_trupdate_wakeup, dummy,
10003                     segvn_update_textrepl_interval);
10004         }
10005 }

10007 static void
10008 segvn_trupdate(void)
10009 {
10010         ulong_t         hash;
10011         svntr_t         *svntrp;
10012         segvn_data_t    *svd;

10014         ASSERT(svntr_hashtab != NULL);

10016         for (hash = 0; hash < svntr_hashtab_sz; hash++) {
10017                 mutex_enter(&svntr_hashtab[hash].tr_lock);
10018                 svntrp = svntr_hashtab[hash].tr_head;
10019                 for (; svntrp != NULL; svntrp = svntrp->tr_next) {
10020                         ASSERT(svntrp->tr_refcnt != 0);
10021                         svd = svntrp->tr_svnhead;
10022                         for (; svd != NULL; svd = svd->svn_trnext) {
10023                                 segvn_trupdate_seg(svd->seg, svd, svntrp,
10024                                     hash);
10025                         }
10026                 }
```

```
10027                   mutex_exit(&svntr_hashtab[hash].tr_lock);
10028           }
10029   }

10031   static void
10032   segvn_trupdate_seg(struct seg *seg,
10033           segvn_data_t *svd,
10034           svntr_t *svntrp,
10035           ulong_t hash)
10036   {
10037           proc_t                  *p;
10038           lgrp_id_t               lgrp_id;
10039           struct as               *as;
10040           size_t                  size;
10041           struct anon_map         *amp;

10043           ASSERT(svd->vp != NULL);
10044           ASSERT(svd->vp == svntrp->tr_vp);
10045           ASSERT(svd->offset == svntrp->tr_off);
10046           ASSERT(svd->offset + seg->s_size == svntrp->tr_eoff);
10047           ASSERT(seg != NULL);
10048           ASSERT(svd->seg == seg);
10049           ASSERT(seg->s_data == (void *)svd);
10050           ASSERT(seg->s_szc == svntrp->tr_szc);
10051           ASSERT(svd->tr_state == SEGVN_TR_ON);
10052           ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
10053           ASSERT(svd->amp != NULL);
10054           ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
10055           ASSERT(svd->tr_policy_info.mem_lgrpid != LGRP_NONE);
10056           ASSERT(svd->tr_policy_info.mem_lgrpid < NLGRPS_MAX);
10057           ASSERT(svntrp->tr_amp[svd->tr_policy_info.mem_lgrpid] == svd->amp);
10058           ASSERT(svntrp->tr_refcnt != 0);
10059           ASSERT(mutex_owned(&svntr_hashtab[hash].tr_lock));

10061           as = seg->s_as;
10062           ASSERT(as != NULL && as != &kas);
10063           p = as->a_proc;
10064           ASSERT(p != NULL);
10065           ASSERT(p->p_tr_lgrpid != LGRP_NONE);
10066           lgrp_id = p->p_t1_lgrpid;
10067           if (lgrp_id == LGRP_NONE) {
10068                   return;
10069           }
10070           ASSERT(lgrp_id < NLGRPS_MAX);
10071           if (svd->tr_policy_info.mem_lgrpid == lgrp_id) {
10072                   return;
10073           }

10075           /*
10076            * Use tryenter locking since we are locking as/seg and svntr hash
10077            * lock in reverse from syncrounous thread order.
10078            */
10079           if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_READER)) {
10080                   SEGVN_TR_ADDSTAT(nolock);
10081                   if (segvn_lgrp_trthr_migrs_snpsht) {
10082                           segvn_lgrp_trthr_migrs_snpsht = 0;
10083                   }
10084                   return;
10085           }
10086           if (!SEGVN_LOCK_TRYENTER(seg->s_as, &svd->lock, RW_WRITER)) {
10087                   AS_LOCK_EXIT(as, &as->a_lock);
10088                   SEGVN_TR_ADDSTAT(nolock);
10089                   if (segvn_lgrp_trthr_migrs_snpsht) {
10090                           segvn_lgrp_trthr_migrs_snpsht = 0;
10091                   }
10092                   return;
```



```
10093           }
10094           size = seg->s_size;
10095           if (svntrp->tr_amp[lgrp_id] == NULL) {
10096                   size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
10097                   if (trmem > segvn_textrepl_max_bytes) {
10098                           SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10099                           AS_LOCK_EXIT(as, &as->a_lock);
10100                          atomic_add_long(&segvn_textrepl_bytes, -size);
10101                          SEGVN_TR_ADDSTAT(normem);
10102                          return;
10103                  }
10104                  if (anon_try_resv_zone(size, NULL) == 0) {
10105                          SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10106                          AS_LOCK_EXIT(as, &as->a_lock);
10107                          atomic_add_long(&segvn_textrepl_bytes, -size);
10108                          SEGVN_TR_ADDSTAT(noanon);
10109                          return;
10110                  }
10111                  amp = anonmap_alloc(size, size, KM_NOSLEEP);
10112                  if (amp == NULL) {
10113                          SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10114                          AS_LOCK_EXIT(as, &as->a_lock);
10115                          atomic_add_long(&segvn_textrepl_bytes, -size);
10116                          anon_unresv_zone(size, NULL);
10117                          SEGVN_TR_ADDSTAT(nokmem);
10118                          return;
10119                  }
10120                  ASSERT(amp->refcnt == 1);
10121                  amp->a_szc = seg->s_szc;
10122                  svntrp->tr_amp[lgrp_id] = amp;
10123          }
10124          /*
10125           * We don't need to drop the bucket lock but here we give other
10126           * threads a chance.  svntr and svd can't be unlinked as long as
10127           * segment lock is held as a writer and AS held as well.  After we
10128           * retake bucket lock we'll continue from where we left. We'll be able
10129           * to reach the end of either list since new entries are always added
10130           * to the beginning of the lists.
10131           */
10132          mutex_exit(&svntr_hashtab[hash].tr_lock);
10133          hat_unload_callback(as->a_hat, seg->s_base, size, 0, NULL);
10134          mutex_enter(&svntr_hashtab[hash].tr_lock);

10136          ASSERT(svd->tr_state == SEGVN_TR_ON);
10137          ASSERT(svd->amp != NULL);
10138          ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
10139          ASSERT(svd->tr_policy_info.mem_lgrpid != lgrp_id);
10140          ASSERT(svd->amp != svntrp->tr_amp[lgrp_id]);

10142          svd->tr_policy_info.mem_lgrpid = lgrp_id;
10143          svd->amp = svntrp->tr_amp[lgrp_id];
10144          p->p_tr_lgrpid = NLGRPS_MAX;
10145          SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10146          AS_LOCK_EXIT(as, &as->a_lock);

10148          ASSERT(svntrp->tr_refcnt != 0);
10149          ASSERT(svd->vp == svntrp->tr_vp);
10150          ASSERT(svd->tr_policy_info.mem_lgrpid == lgrp_id);
10151          ASSERT(svd->amp != NULL && svd->amp == svntrp->tr_amp[lgrp_id]);
10152          ASSERT(svd->seg == seg);
10153          ASSERT(svd->tr_state == SEGVN_TR_ON);

10155          SEGVN_TR_ADDSTAT(asyncrepl);
10156  }
```