

new/usr/src/uts/common/vm/pvn.h

```
*****
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>
*****
1 /*
2 * CDDL HEADER START
3 *
4 * The contents of this file are subject to the terms of the
5 * Common Development and Distribution License (the "License").
6 * You may not use this file except in compliance with the License.
7 *
8 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /*
26 * Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
27 /* All Rights Reserved */
28 /*
29 * University Copyright- Copyright (c) 1982, 1986, 1988
30 * The Regents of the University of California
31 * All Rights Reserved
32 *
33 * University Acknowledgment- Portions of this document are derived from
34 * software developed by the University of California, Berkeley, and its
35 * contributors.
36 */
37 */
38 #ifndef _VM_PVN_H
39 #define _VM_PVN_H
40
41 #include <sys/buf.h>
42 #include <vm/seg.h>
43
44 #ifdef __cplusplus
45 extern "C" {
46#endif
47
48 #ifdef _KERNEL
49
50 /*
51 * VM - paged vnode.
52 *
53 * The VM system manages memory as a cache of paged vnodes.
54 * This file describes the interfaces to common subroutines
55 * used to help implement the VM/file system routines.
56 */
57
58 struct page *pvn_read_kluster(struct vnode *vp, u_offset_t off,
59 struct seg *seg, caddr_t addr, u_offset_t *offp,
```

1

new/usr/src/uts/common/vm/pvn.h

```
61 size_t *lennp, u_offset_t vp_off, size_t vp_len,
62 int isra);
63 struct page *pvn_write_kluster(struct vnode *vp, struct page *pp,
64 u_offset_t *offp, size_t *lennp, u_offset_t vp_off,
65 size_t vp_len, int flags);
66 void pvn_read_done(struct page *plist, int flags);
67 void pvn_write_done(struct page *plist, int flags);
68 void pvn_io_done(struct page *plist);
69 int pvn_vp_list_dirty(struct vnode *vp, u_offset_t off,
70 int (*putapage)(vnode_t *, struct page *, u_offset_t *,
71 size_t *, int, cred_t *),
72 int flags, struct cred *cred);
73 void pvn_vp_list_setdirty(vnode_t *vp, int (*page_check)(page_t *));
74 int pvn_getdirty(struct page *pp, int flags);
75 void pvn_vpzero(struct vnode *vp, u_offset_t vplen, size_t zbytes);
76 int pvn_getpages(
77 int (*getpage)(vnode_t *, u_offset_t, size_t, uint_t *,
78 struct page *[], size_t, struct seg *,
79 caddr_t, enum seg_rw, cred_t *),
80 struct vnode *vp, u_offset_t off, size_t len,
81 uint_t *protp, struct page **pl, size_t plsz,
82 struct seg *seg, caddr_t addr, enum seg_rw rw,
83 struct cred *cred);
84 void pvn_plist_init(struct page *pp, struct page **pl, size_t plsz,
85 u_offset_t off, size_t io_len, enum seg_rw rw);
86 void pvn_init(void);
87 /*
88 * The value is put in p_hash to identify marker pages. It is safe to
89 * test p_hash ==(!=) PVN_VPLIST_HASH_TAG even without holding p_selock.
90 */
91 #define PVN_VPLIST_HASH_TAG ((page_t *)-1)
92
93 /*
94 * When requesting pages from the getpage routines, pvn_getpages will
95 * allocate space to return PVN_GETPAGE_NUM pages which map PVN_GETPAGE_SZ
96 * worth of bytes. These numbers are chosen to be the minimum of the max's
97 * given in terms of bytes and pages.
98 */
99
100 #define PVN_MAX_GETPAGE_SZ 0x10000 /* getpage size limit */
101 #define PVN_MAX_GETPAGE_NUM 0x8 /* getpage page limit */
102
103 #if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
104
105 #define PVN_GETPAGE_SZ ptob(PVN_MAX_GETPAGE_NUM)
106 #define PVN_GETPAGE_NUM btop(PVN_MAX_GETPAGE_SZ)
107
108 #else
109
110 #define PVN_GETPAGE_SZ PVN_MAX_GETPAGE_SZ
111 #define PVN_GETPAGE_NUM PVN_MAX_GETPAGE_NUM
112
113 #endif
114
115 #endif /* _KERNEL */
116
117 #ifdef __cplusplus
118 }
119
120 #endif /* unchanged_portion_omitted_ */

2
```

```
new/usr/src/uts/common/vm/seg_vn.c
```

```
*****
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>
*****
1 /*
2 * CDDL HEADER START
3 *
4 * The contents of this file are subject to the terms of the
5 * Common Development and Distribution License (the "License").
6 * You may not use this file except in compliance with the License.
7 *
8 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
24 #endif /* ! codereview */
25 */
26 /*
27 /* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
28 /* All Rights Reserved */
29 */
30 /*
31 * University Copyright- Copyright (c) 1982, 1986, 1988
32 * The Regents of the University of California
33 * All Rights Reserved
34 *
35 * University Acknowledgment- Portions of this document are derived from
36 * software developed by the University of California, Berkeley, and its
37 * contributors.
38 */
39 /*
40 * VM - shared or copy-on-write from a vnode/anonymous memory.
41 */
42 */

43 #include <sys/types.h>
44 #include <sys/param.h>
45 #include <sys/t_lock.h>
46 #include <sys/errno.h>
47 #include <sys/systm.h>
48 #include <sys/mman.h>
49 #include <sys/debug.h>
50 #include <sys/cred.h>
51 #include <sys/vmsystm.h>
52 #include <sys/tunable.h>
53 #include <sys/bitmap.h>
54 #include <sys/swap.h>
55 #include <sys/kmem.h>
56 #include <sys/sysmacros.h>
57 #include <sys/vtrace.h>
58 #include <sys/cmn_err.h>
59 #include <sys/callb.h>
60 
```

1

```
new/usr/src/uts/common/vm/seg_vn.c
```

```
61 #include <sys/vm.h>
62 #include <sys/dumpdr.h>
63 #include <sys/lgrp.h>
64
65 #include <vm/hat.h>
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/seg_vn.h>
69 #include <vm/pvn.h>
70 #include <vm/anon.h>
71 #include <vm/page.h>
72 #include <vm/vpage.h>
73 #include <sys/proc.h>
74 #include <sys/task.h>
75 #include <sys/project.h>
76 #include <sys/zone.h>
77 #include <sys/shm_impl.h>
78
79 /*
80 * segvn_fault needs a temporary page list array. To avoid calling kmem all
81 * the time, it creates a small (PVN_MAX_GETPAGE_NUM entry) array and uses it if
82 * it can. In the rare case when this page list is not large enough, it
83 * goes and gets a large enough array from kmem.
84 *
85 * This small page list array covers either 8 pages or 64kB worth of pages -
86 * whichever is smaller.
87 */
88 #define PVN_MAX_GETPAGE_SZ      0x10000
89 #define PVN_MAX_GETPAGE_NUM     0x8
90
91 #if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
92 #define PVN_GETPAGE_SZ ptob(PVN_MAX_GETPAGE_NUM)
93 #define PVN_GETPAGE_NUM PVN_MAX_GETPAGE_NUM
94 #else
95 #define PVN_GETPAGE_SZ PVN_MAX_GETPAGE_SZ
96 #define PVN_GETPAGE_NUM btop(PVN_MAX_GETPAGE_SZ)
97 #endif
98
99 #endif /* ! codereview */
100 /*
101 * Private seg op routines.
102 */
103 static int      segvn_dup(struct seg *seg, struct seg *newseg);
104 static int      segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
105 static void     segvn_free(struct seg *seg);
106 static faultcode_t segvn_fault(struct hat *hat, struct seg *seg,
107                                caddr_t addr, size_t len, enum fault_type type,
108                                enum seg_rw rw);
109 static faultcode_t segvn_faulta(struct seg *seg, caddr_t addr);
110 static int      segvn_setprot(struct seg *seg, caddr_t addr,
111                               size_t len, uint_t prot);
112 static int      segvn_checkprot(struct seg *seg, caddr_t addr,
113                                 size_t len, uint_t prot);
114 static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
115 static size_t    segvn_swapout(struct seg *seg, *seg);
116 static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
117                           int attr, uint_t flags);
118 static size_t    segvn_incore(struct seg *seg, caddr_t addr, size_t len,
119                               char *vec);
120 static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
121                           int attr, int op, ulong_t *lockmap, size_t pos);
122 static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
123                               uint_t *prot);
124 static u_offset_t segvn_getoffset(struct seg *seg, caddr_t addr);
125 static int      segvn_gettime(struct seg *seg, caddr_t addr);
126 static int      segvn_getvp(struct seg *seg, caddr_t addr,
```

2

```

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_gettime,
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_set vnode_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 segvnmstats_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 } segvnmstats;
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     }
260 /*ARGSUSED*/
261 static int
262 segvn_cache_constructor(void *buf, void *cdrarg, int kmflags)
263 {
264     struct segvn_data *svd = buf;
265
266     rw_init(&svd->lock, NULL, RW_DEFAULT, NULL);
267     mutex_init(&svd->segfree_syncmtx, NULL, MUTEX_DEFAULT, NULL);
268     svd->svn_trnext = svd->svn_trprev = NULL;
269     return (0);
270 }
271 */
272 /*ARGSUSED1*/
273 static void
274 segvn_cache_destructor(void *buf, void *cdrarg)
275 {
276     struct segvn_data *svd = buf;
277
278     rw_destroy(&svd->lock);
279     mutex_destroy(&svd->segfree_syncmtx);
280 }
281 */
282 /*ARGSUSED*/
283 static int
284 svntr_cache_constructor(void *buf, void *cdrarg, int kmflags)
285 {
286     bzero(buf, sizeof(svntr_t));
287     return (0);
288 }
289 */
290 /*
291 * Patching this variable to non-zero allows the system to run with
292 * stacks marked as "not executable". It's a bit of a kludge, but is
293 * provided as a tweakable for platforms that export those ABIs
294 * (e.g. sparc V8) that have executable stacks enabled by default.
295 * There are also some restrictions for platforms that don't actually
296 * implement 'noexec' protections.
297 *
298 * Once enabled, the system is (therefore) unable to provide a fully
299 * ABI-compliant execution environment, though practically speaking,
300 * most everything works. The exceptions are generally some interpreters
301 * and debuggers that create executable code on the stack and jump
302 * into it (without explicitly mprotecting the address range to include
303 * PROT_EXEC).
304 *
305 * One important class of applications that are disabled are those
306 * that have been transformed into malicious agents using one of the
307 * numerous "buffer overflow" attacks. See 4007890.
308 */
309 int noexec_user_stack = 0;
310 int noexec_user_stack_log = 1;
311
312 int segvn_lpg_disable = 0;
313 uint_t segvn_maxpgszc = 0;
314
315 ulong_t segvn_vmpss_clrszc_cnt;
316 ulong_t segvn_vmpss_clrszc_err;
317 ulong_t segvn_fltvnpages_clrszc_cnt;
318 ulong_t segvn_fltvnpages_clrszc_err;
319 ulong_t segvn_setpgsz_align_err;
320 ulong_t segvn_setpgsz_anon_align_err;
321 ulong_t segvn_setpgsz_getattr_err;
322 ulong_t segvn_setpgsz_eof_err;
323 ulong_t segvn_faultvnpmpss_align_err1;
324 ulong_t segvn_faultvnpmpss_align_err1;

```

```

325 ulong_t segvn_faultvnpmpss_align_err2;
326 ulong_t segvn_faultvnpmpss_align_err3;
327 ulong_t segvn_faultvnpmpss_align_err4;
328 ulong_t segvn_faultvnpmpss_align_err5;
329 ulong_t segvn_vmpss_pageio_deadlk_err;
330
331 int segvn_use_regions = 1;
332 */
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 rebinding segvn_trsync_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 */
378
379 #define SEGVN_TEXTREPL_MAXBYTES_FACTOR (20)
380 size_t segvn_textrepl_max_bytes_factor = SEGVN_TEXTREPL_MAXBYTES_FACTOR;
381
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_trsync_sem;
387
388 int segvn_disable_textrepl = 1;
389 size_t textrepl_size_thresh = (size_t)-1;
390 size_t segvn_textrepl_bytes = 0;

```

```

391 size_t segvn_textrepl_max_bytes = 0;
392 clock_t segvn_update_textrepl_interval = 0;
393 int segvn_update_tr_time = 10;
394 int segvn_disable_textrepl_update = 0;

396 static void segvn_textrepl(struct seg *);
397 static void segvn_textunrepl(struct seg *, int);
398 static void segvn_inval_trcache(vnode_t *);
399 static void segvn_trasync_thread(void);
400 static void segvn_trupdate_wakeup(void *);
401 static void segvn_trupdate(void);
402 static void segvn_trupdate_seg(struct seg *, segvn_data_t *, svntr_t *,
403     ulong_t);

405 /*
406  * Initialize segvn data structures
407  */
408 void segvn_init(void)
409 {
410     uint_t maxszc;
411     uint_t szc;
412     size_t pgsz;

415     segvn_cache = kmem_cache_create("segvn_cache",
416         sizeof (struct segvn_data), 0,
417         segvn_cache_constructor, segvn_cache_destructor, NULL,
418         NULL, NULL, 0);

420     if (segvn_lpg_disable == 0) {
421         szc = maxszc = page_num_pagesizes() - 1;
422         if (szc == 0) {
423             segvn_lpg_disable = 1;
424         }
425         if (page_get_pagesize(0) != PAGESIZE) {
426             panic("segvn_init: bad szc %d", szc);
427             /*NOTREACHED*/
428         }
429         while (szc != 0) {
430             pgsz = page_get_pagesize(szc);
431             if (pgsz <= PAGESIZE || !IS_P2ALIGNED(pgsz, pgsz)) {
432                 panic("segvn_init: bad szc %d", szc);
433                 /*NOTREACHED*/
434             }
435             szc--;
436         }
437         if (segvn_maxpgszc == 0 || segvn_maxpgszc > maxszc)
438             segvn_maxpgszc = maxszc;
439     }

441     if (segvn_maxpgszc) {
442         segvn_szc_cache = (struct kmem_cache **)kmem_alloc(
443             (segvn_maxpgszc + 1) * sizeof (struct kmem_cache *),
444             KM_SLEEP);
445     }

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

450         (void) sprintf(str, "segvn_szc_cache%d", szc);
451         segvn_szc_cache[szc] = kmem_cache_create(str,
452             page_get_pagecnt(szc) * sizeof (page_t *), 0,
453             NULL, NULL, NULL, NULL, KMC_NODEBUG);
454     }

```

```

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

460     /*
461      * For now shared regions and text replication segvn support
462      * are mutually exclusive. This is acceptable because
463      * currently significant benefit from text replication was
464      * only observed on AMD64 NUMA platforms (due to relatively
465      * small L2$ size) and currently we don't support shared
466      * regions on x86.
467      */
468     if (segvn_use_regions && !segvn_disable_textrepl) {
469         segvn_disable_textrepl = 1;
470     }

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

478         svntr_cache = kmem_cache_create("svntr_cache",
479             sizeof (svntr_t), 0, svntr_cache_constructor, NULL,
480             NULL, NULL, NULL, 0);
481         svntr_hashtab = kmem_zalloc(hsz, KM_SLEEP);
482         for (i = 0; i < svntr_hashtab_sz; i++) {
483             mutex_init(&svntr_hashtab[i].tr_lock, NULL,
484                         MUXTEX_DEFAULT, NULL);
485         }
486         segvn_textrepl_max_bytes = ptob(phymem) /
487             segvn_textrepl_max_bytes_factor;
488         segvn_textrepl_stats = kmem_zalloc(NCPU *
489             sizeof (svntr_stats_t), KM_SLEEP);
490         sema_init(&segvn_trasync_sem, 0, NULL, SEMA_DEFAULT, NULL);
491         (void) thread_create(NULL, 0, segvn_trasync_thread,
492             NULL, 0, &p0, TS_RUN, minclsyspri);
493     }
494 #endif

496     if (!ISP2(segvn_pclock_comb_balign) ||
497         segvn_pclock_comb_balign < PAGESIZE) {
498         segvn_pclock_comb_balign = 1UL << 16; /* 64K */
499     }
500     segvn_pclock_comb_bshift = highbit(segvn_pclock_comb_balign) - 1;
501     segvn_pclock_comb_palign = bttop(segvn_pclock_comb_balign);
502 }

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

507 static void segvn_set vnode_mpss(vnode_t *vp)
508 {
509     int err;
510
512     ASSERT(vp->v_mpssdata == NULL ||
513         vp->v_mpssdata == SEGVN_PAGEIO ||
514         vp->v_mpssdata == SEGVN_NOPAGEIO);
516
516     if (vp->v_mpssdata == NULL) {
517         if (vn_mpss_usepageio(vp)) {
518             err = VOP_PAGEIO(vp, (page_t *)NULL,
519                             (u_offset_t)0, 0, 0, CRED(), NULL);
520         } else {
521             err = ENOSYS;
522         }
523     }

```

```

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

```

```

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                 } else if (a->amp != NULL) {
603                     pgcnt_t anum = btopr(a->offset);
604                     pgcnt_t pgcnt = page_get_pagecnt(a->szc);
605                     if (!IS_P2ALIGNED(anum, pgcnt)) {
606                         a->szc = 0;
607                     }
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         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         }
656     }

```

`new/usr/src/uts/common/vm/seg_vn.c`

11

```

555         crfree(cred);
556         if (!use_rgn) {
557             hat_unload(seg->s_as->a_hat, seg->s_base,
558                         seg->s_size, HAT_UNLOAD_UNMAP);
559         }
560     }
561 }
562 */
563 * svntr_hashtab will be NULL if we support shared regions.
564 */
565 trok = ((a->flags & MAP_TEXT) &&
566         (seg->s_size > textrepl_size_thresh ||

567         (a->flags & _MAP_TEXTREPL)) &&
568         lgrp_optimizations() && svntr_hashtab != NULL &&
569         a->type == MAP_PRIVATE && swresv == 0 &&
570         !(a->flags & MAP_NORESERVE) &&
571         seg->s_as != &kas && a->vp->v_type == VREG);

572     ASSERT(!trok || !use_rgn);
573 }

574 /*
575 * MAP_NORESERVE mappings don't count towards the VSZ of a process
576 * until we fault the pages in.
577 */
578 if ((a->vp == NULL || a->vp->v_type != VREG) &&
579     a->flags & MAP_NORESERVE) {
580     seg->s_as->a_resvsize -= seg->s_size;
581 }

582 /*
583 * If more than one segment in the address space, and they're adjacent
584 * virtually, try to concatenate them. Don't concatenate if an
585 * explicit anon_map structure was supplied (e.g., SystemV shared
586 * memory) or if we'll use text replication for this segment.
587 */
588 if (a->amp == NULL && !use_rgn && !trok) {
589     struct seg *pseg, *nseg;
590     struct segvn_data *psvd, *nsvd;
591     lgrp_mem_policy_t ppolicy, npolicy;
592     uint_t lgrp_mem_policy_flags = 0;
593     extern lgrp_mem_policy_t lgrp_mem_default_policy;

594     /*
595      * Memory policy flags (lgrp_mem_policy_flags) is valid when
596      * extending stack/heap segments.
597      */
598     if ((a->vp == NULL) && (a->type == MAP_PRIVATE) &&
599         !(a->flags & MAP_NORESERVE) && (seg->s_as != &kas)) {
600         lgrp_mem_policy_flags = a->lgrp_mem_policy_flags;
601     } else {
602         /*
603          * Get policy when not extending it from another segment
604          */
605         mpolicy = lgrp_mem_policy_default(seg->s_size, a->type);
606     }

607     /*
608      * First, try to concatenate the previous and new segments
609      */
610     pseg = AS_SEGPREV(seg->s_as, seg);
611     if (pseg != NULL &&
612         pseg->s_base + pseg->s_size == seg->s_base &&
613         pseg->s_ops == &segvn_ops) {
614         /*
615          * Get memory allocation policy from previous segment.
616          */
617         seg->s_size += pseg->s_size;
618         seg->s_ops = pseg->s_ops;
619     }
620 }

```

`new/usr/src/uts/common/vm/seg_vn.`

```

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

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

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

```

```

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 if (a->vp != NULL) {
804     VN_HOLD(a->vp);
805     if (a->type == MAP_SHARED)
806         lgrp_shm_policy_init(NULL, a->vp);
807 }
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 & VVMEEXEC) && (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     } else {
862         /*
863          * Private mapping (with or without a vp).
864          * Allocate anon_map when needed.
865          */
866         svd->swresv = swresv;
867     }
868 } else {
869     pgcnt_t anon_num;
870
871     /*
872      * Mapping to an existing anon_map structure without a vp.
873      * For now we will insure that the segment size isn't larger
874      * than the size - offset gives us. Later on we may wish to
875      * have the anon array dynamically allocated itself so that
876      * we don't always have to allocate all the anon pointer slots.
877      * This of course involves adding extra code to check that we
878      * aren't trying to use an anon pointer slot beyond the end
879      * of the currently allocated anon array.
880      */
881     if ((amp->size - a->offset) < seg->s_size) {
882         panic("segvn_create anon_map size");
883         /*NOTREACHED*/
884     }
885
886     anon_num = btopr(a->offset);
887
888     if (a->type == MAP_SHARED) {
889         /*
890          * SHARED mapping to a given anon_map.
891          */
892         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
893         amp->refcnt++;
894         if (a->szc > amp->a_szc) {
895             amp->a_szc = a->szc;
896         }
897         ANON_LOCK_EXIT(&amp->a_rwlock);
898         svd->anon_index = anon_num;
899         svd->swresv = 0;
900     } else {
901         /*
902          * PRIVATE mapping to a given anon_map.
903          * Make sure that all the needed anon
904          * structures are created (so that we will
905          * share the underlying pages if nothing
906          * is written by this mapping) and then
907          * duplicate the anon array as is done
908          * when a privately mapped segment is dup'ed.
909          */
910         struct anon *ap;
911         caddr_t addr;
912         caddr_t eaddr;
913         ulong_t anon_idx;
914         int hat_flag = HAT_LOAD;
915
916         if (svd->flags & MAP_TEXT) {
917             hat_flag |= HAT_LOAD_TEXT;
918

```

```

919         }
920
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;
925
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;
932
933         for (anon_idx = anon_num, addr = seg->s_base;
934              addr < eaddr; addr += PAGESIZE, anon_idx++) {
935             page_t *pp;
936
937             if ((ap = anon_get_ptr(amp->ahp,
938                                   anon_idx)) != NULL)
939               continue;
940
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             }
951
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);
960
961             ASSERT(seg->s_szc == 0);
962             ASSERT(!IS_VMODSORT(pp->p_vnode));
963
964             ASSERT(use_rgn == 0);
965             hat_memload(seg->s_as->a_hat, addr, pp,
966                         svd->prot & ~PROT_WRITE, hat_flag);
967
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
976     /*
977      * Set default memory allocation policy for segment
978      *
979      * Always set policy for private memory at least for initialization
980      * even if this is a shared memory segment
981      */
982     (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);
988
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->prot,
994                                         (uchar_t)seg->s_szc, segvn_hat_rgn_unload_callback,
995                                         HAT_REGION_TEXT);
996         }
997
998         ASSERT(!trok || !(svd->prot & PROT_WRITE));
999         svd->tr_state = trok ? SEGVN_TR_INIT : SEGVN_TR_OFF;
1000
1001     return (0);
1002 }
1003
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;
1021
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);
1025
1026   if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie) ||
1027       HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1028     return (-1);
1029   }
1030
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
1041
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   }
1049
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     * Fail early if we're not supposed to concatenate
1058     * segments with non NULL amp.
1059     */
1060     if (amp_cat == 0 && (amp1 != NULL || amp2 != NULL)) {
1061         return (-1);
1062     }
1063
1064     if (svd1->vp == NULL && svd1->type == MAP_SHARED) {
1065         if (amp1 != amp2) {
1066             return (-1);
1067         }
1068         if (amp1 != NULL && svd1->anon_index + btop(seg1->s_size) !=
1069             svd2->anon_index) {
1070             return (-1);
1071         }
1072         ASSERT(amp1 == NULL || amp1->refcnt >= 2);
1073     }
1074
1075     /*
1076     * If either seg has vpages, create a new merged vpage array.
1077     */
1078     if (vpage1 != NULL || vpage2 != NULL) {
1079         struct vpage *vp, *evp;
1080
1081         npages1 = seg_pages(seg1);
1082         npages2 = seg_pages(seg2);
1083         nvpsize = vpgtob(npages1 + npages2);
1084
1085         if ((nvpage = kmem_zalloc(nvpsize, KM_NOSLEEP)) == NULL) {
1086             return (-2);
1087         }
1088
1089         if (vpage1 != NULL) {
1090             bcopy(vpage1, nvpage, vpgtob(npages1));
1091         } else {
1092             evp = nvpage + npages1;
1093             for (vp = nvpage; vp < evp; vp++) {
1094                 VPP_SETPROT(vp, svd1->prot);
1095                 VPP_SETADVICE(vp, svd1->advice);
1096             }
1097         }
1098
1099         if (vpage2 != NULL) {
1100             bcopy(vpage2, nvpage + npages1, vpgtob(npages2));
1101         } else {
1102             evp = nvpage + npages1 + npages2;
1103             for (vp = nvpage + npages1; vp < evp; vp++) {
1104                 VPP_SETPROT(vp, svd2->prot);
1105                 VPP_SETADVICE(vp, svd2->advice);
1106             }
1107         }
1108
1109         if (svd2->pageswap && (!svd1->pageswap && svd1->swresv)) {
1110             ASSERT(svd1->swresv == seg1->s_size);
1111             ASSERT(!(svd1->flags & MAP_NORESERVE));
1112             ASSERT(!(svd2->flags & MAP_NORESERVE));
1113             evp = nvpage + npages1;
1114             for (vp = nvpage; vp < evp; vp++) {
1115                 VPP_SETSWAPRES(vp);
1116             }
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 merging 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         asize = seg1->s_size + seg2->s_size;
1153         if ((nahp = anon_create(btop(asize), ANON_NOSLEEP)) == NULL) {
1154             if (nvpage != NULL) {
1155                 kmem_free(nvpage, nvpsize);
1156             }
1157             return (-2);
1158         }
1159         if (amp1 != NULL) {
1160             /*
1161             * XXX anon rwlock is not really needed because
1162             * this is a private segment and we are writers.
1163             */
1164             ANON_LOCK_ENTER(&amp1->a_rwlock, RW_WRITER);
1165             ASSERT(amp1->refcnt == 1);
1166             if (anon_copy_ptr(amp1->ahp, svd1->anon_index,
1167                               nahp, 0, btop(seg1->s_size), ANON_NOSLEEP)) {
1168                 anon_release(nahp, btop(asize));
1169                 ANON_LOCK_EXIT(&amp1->a_rwlock);
1170                 if (nvpage != NULL) {
1171                     kmem_free(nvpage, nvpsize);
1172                 }
1173             }
1174             return (-2);
1175         }
1176         if (amp2 != NULL) {
1177             ANON_LOCK_ENTER(&amp2->a_rwlock, RW_WRITER);
1178             ASSERT(amp2->refcnt == 1);
1179             if (anon_copy_ptr(amp2->ahp, svd2->anon_index,
1180                               nahp, btop(seg2->s_size), btop(seg2->s_size),
1181                               ANON_NOSLEEP)) {
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
20100
20101
20102
20103
20104
20105
20106
20107
20108
20109
20110
20111
20112
20113
20114
20115
20116
20117
20118
20119
20120
20121
20122
20123
20124
20125
20126
20127
20128
20129
20130
20131
20132
20133
20134
20135
20136
20137
20138
20139
20140
20141
20142
20143
20144
20145
20146
20147
20148
20149
20150
20151
20152
20153
20154
20155
20156
20157
20158
20159
20160
20161
20162
20163
20164
20165
20166
20167
20168
20169
20170
20171
20172
20173
20174
20175
20176
20177
20178
20179
20180
20181
20182
20183
20184
20185
20186
20187
20188
20189
20190
20191
20192
20193
20194
20195
20196
20197
20198
20199
20200
20201
20202
20203
20204
20205
20206
20207
20208
20209
20210
20211
20212
20213
20214
20215
20216
20217
20218
20219
20220
20221
20222
20223
20224
20225
20226
20227
20228
20229
20230
20231
20232
20233
20234
20235
20236
20237
20238
20239
20240
20241
20242
20243
20244
20245
20246
20247
20248
20249
20250
20251
20252
20253
20254
20255
20256
20257
20258
20259
202510
202511
202512
202513
202514
202515
202516
202517
202518
202519
202520
202521
202522
202523
202524
202525
202526
202527
202528
202529
202530
202531
202532
202533
202534
202535
202536
202537
202538
202539
202540
202541
202542
202543
202544
202545
202546
202547
202548
202549
202550
202551
202552
202553
202554
202555
202556
202557
202558
202559
202560
202561
202562
202563
202564
202565
202566
202567
202568
202569
202570
202571
202572
202573
202574
202575
202576
202577
202578
202579
202580
202581
202582
202583
202584
202585
202586
202587
202588
202589
202590
202591
202592
202593
202594
202595
202596
202597
202598
202599
2025100
2025101
2025102
2025103
2025104
2025105
2025106
2025107
2025108
2025109
2025110
2025111
2025112
2025113
2025114
2025115
2025116
2025117
2025118
2025119
2025120
2025121
2025122
2025123
2025124
2025125
2025126
2025127
2025128
2025129
2025130
2025131
2025132
2025133
2025134
2025135
2025136
2025137
2025138
2025139
2025140
2025141
2025142
2025143
2025144
2025145
2025146
2025147
2025148
2025149
2025150
2025151
2025152
2025153
2025154
2025155
2025156
2025157
2025158
2025159
20251510
20251511
20251512
20251513
20251514
20251515
20251516
20251517
20251518
20251519
20251520
20251521
20251522
20251523
20251524
20251525
20251526
20251527
20251528
20251529
202515210
202515211
202515212
202515213
202515214
202515215
202515216
202515217
202515218
202515219
202515220
202515221
202515222
202515223
202515224
202515225
202515226
202515227
202515228
202515229
202515230
202515231
202515232
202515233
202515234
202515235
202515236
202515237
202515238
202515239
202515240
202515241
202515242
202515243
202515244
202515245
202515246
202515247
202515248
202515249
202515250
202515251
202515252
202515253
202515254
202515255
202515256
202515257
202515258
202515259
202515260
202515261
202515262
202515263
202515264
202515265
202515266
202515267
202515268
202515269
202515270
202515271
202515272
202515273
202515274
202515275
202515276
202515277
202515278
202515279
202515280
202515281
202515282
202515283
202515284
202515285
202515286
202515287
202515288
202515289
202515290
202515291
202515292
202515293
202515294
202515295
202515296
202515297
202515298
202515299
2025152100
2025152101
2025152102
2025152103
2025152104
2025152105
2025152106
2025152107
2025152108
2025152109
2025152110
2025152111
2025152112
2025152113
2025152114
2025152115
2025152116
2025152117
2025152118
2025152119
2025152120
2025152121
2025152122
2025152123
2025152124
2025152125
2025152126
2025152127
2025152128
2025152129
2025152130
2025152131
2025152132
2025152133
2025152134
2025152135
2025152136
2025152137
2025152138
2025152139
2025152140
2025152141
2025152142
2025152143
2025152144
2025152145
2025152146
2025152147
2025152148
2025152149
2025152150
2025152151
2025152152
2025152153
2025152154
2025152155
2025152156
2025152157
2025152158
2025152159
2025152160
2025152161
2025152162
2025152163
2025152164
2025152165
2025152166
2025152167
2025152168
2025152169
2025152170
2025152171
2025152172
2025152173
2025152174
2025152175
2025152176
2025152177
2025152178
2025152179
2025152180
2025152181
2025152182
2025152183
2025152184
2025152185
2025152186
2025152187
2025152188
2025152189
2025152190
2025152191
2025152192
2025152193
2025152194
2025152195
2025152196
2025152197
2025152198
2025152199
2025152200
2025152201
2025152202
2025152203
2025152204
2025152205
2025152206
2025152207
2025152208
2025152209
2025152210
2025152211
2025152212
2025152213
2025152214
2025152215
2025152216
2025152217
2025152218
2025152219
2025152220
2025152221
2025152222
2025152223
2025152224
2025152225
2025152226
2025152227
2025152228
2025152229
2025152230
2025152231
2025152232
2025152233
2025152234
2025152235
2025152236
2025152237
2025152238
2025152239
2025152240
2025152241
2025152242
2025
```

```

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     if (amp1 != NULL) {
1194         namp = amp1;
1195         anon_release(amp1->ahp, btop(amp1->size));
1196     }
1197     if (amp2 != NULL) {
1198         if (namp == NULL) {
1199             ASSERT(amp1 == NULL);
1200             namp = amp2;
1201             anon_release(amp2->ahp, btop(amp2->size));
1202         } else {
1203             amp2->refcnt--;
1204             ANON_LOCK_EXIT(&amp2->a_rwlock);
1205             anonmap_free(amp2);
1206         }
1207         svd2->amp = NULL; /* needed for seg_free */
1208     }
1209     namp->ahp = nahp;
1210     namp->size = asize;
1211     svd1->amp = namp;
1212     svd1->anon_index = 0;
1213     ANON_LOCK_EXIT(&namp->a_rwlock);
1214 }
1215 /*
1216 * Now free the old vpage structures.
1217 */
1218 if (nvpage != NULL) {
1219     if (vpage1 != NULL) {
1220         kmem_free(vpage1, vpgtob(npages1));
1221     }
1222     if (vpage2 != NULL) {
1223         svd2->vpage = NULL;
1224         kmem_free(vpage2, vpgtob(npages2));
1225     }
1226     if (svd2->pageprot) {
1227         svd1->pageprot = 1;
1228     }
1229     if (svd2->pageadvice) {
1230         svd1->pageadvice = 1;
1231     }
1232     if (svd2->pageswap) {
1233         svd1->pageswap = 1;
1234     }
1235     svd1->vpage = nvpage;
1236 }
1237 */
1238 /* all looks ok, merge segments */
1239 svd1->swresv += svd2->swresv;
1240 svd2->swresv = 0; /* so seg_free doesn't release swap space */
1241 size = seg2->s_size;
1242 seg_free(seg2);
1243 seg1->s_size += size;
1244 return (0);
1245 }
1246 */
1247 */

```

```

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;
1263
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));
1269
1270     if (HAT_IS_REGION_COOKIE_VALID(svd1->rcookie)) {
1271         return (-1);
1272     }
1273
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);
1281
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);
1286
1287     if (svd1->tr_state != SEGVN_TR_OFF) {
1288         return (-1);
1289     }
1290
1291     amp1 = svd1->amp;
1292     if (amp1) {
1293         pgcnt_t newpgs;
1294
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(&ampl->a_rwlock, RW_WRITER);
1316     if (ampl->refcnt > 1) {
1317         ANON_LOCK_EXIT(&ampl->a_rwlock);
1318         return (-1);
1319     }
1320     newpgs = anon_grow(ampl->ahp, &svd1->anon_index,
1321                         btop(seg1->s_size), btop(seg2->s_size), ANON_NOSLEEP);
1322
1323     if (newpgs == 0) {
1324         ANON_LOCK_EXIT(&ampl->a_rwlock);
1325         return (-1);
1326     }
1327     ampl->size = ptob(newpgs);
1328     ANON_LOCK_EXIT(&ampl->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;
1340
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->fflags & 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     ASSERT(svd1->vpage != NULL || svd1->pageswap == 0);
1354     size = seg2->s_size;
1355     seg_free(seg2);
1356     seg1->s_size += size;
1357     svd1->swresv += swresv;
1358     if (svd1->pageprot && (a->prot & PROT_WRITE) &&
1359         svd1->type == MAP_SHARED && svd1->vp != NULL &&
1360         (svd1->vp->v_flag & VVEXEC)) {
1361         ASSERT(vn_is_mapped(svd1->vp, V_WRITE));
1362         segvn_inval_trcache(svd1->vp);
1363     }
1364 }
1365 return (0);
1366 }

1367 */
1368 /* Extend the next segment (seg2) to include the
1369 * new segment (seg1 + a), if possible.
1370 * Return 0 on success.
1371 */
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;
1384
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));
1390
1391     if (HAT_IS_REGION_COOKIE_VALID(svd2->rcookie)) {
1392         return (-1);
1393     }
1394
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);
1406
1407     if (svd2->tr_state != SEGVN_TR_OFF) {
1408         return (-1);
1409     }
1410
1411     amp2 = svd2->amp;
1412     if (amp2) {
1413         pgcnt_t newpgs;
1414
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);
1429
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);
1438
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;
1461
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     ASSERT(svd2->vpage != NULL || svd2->pageswap == 0);
1475     size = seg1->s_size;
1476     seg_free(seg1);
1477     seg2->s_size += size;
1478     seg2->s_base -= size;
1479     svd2->offset -= size;
1480     svd2->swresv += swresv;
1481     if (svd2->pageprot && (a->prot & PROT_WRITE) &&
1482         svd2->type == MAP_SHARED && svd2->vp != NULL &&
1483         (svd2->vp->v_flag & VMEXEC)) {
1484         ASSERT(vn_is_mapped(svd2->vp, V_WRITE));
1485         segvn_inval_trcache(svd2->vp);
1486     }
1487     return (0);
1488 }
1489
1490 static int
1491 segvn_dup(struct seg *seg, struct seg *newseg)
1492 {
1493     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
1494     struct segvn_data *newsyd;
1495     pgcnt_t npages = seg_pages(seg);
1496     int error = 0;
1497     uint_t prot;
1498     size_t len;
1499     struct anon_map *amp;
1500
1501     ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
1502     ASSERT(newseg->s_as->a_proc->p_parent == curproc);
1503
1504     /*
1505      * If segment has anon reserved, reserve more for the new seg.
1506      * For a MAP_NORESERVE segment swresv will be a count of all the
1507      * allocated anon slots; thus we reserve for the child as many slots
1508      * as the parent has allocated. This semantic prevents the child or
1509      * parent from dieing during a copy-on-write fault caused by trying
1510      * to write a shared pre-existing anon page.
1511     */

```

```

1513     if ((len = svd->swresv) != 0) {
1514         if (anon_resv(svd->swresv) == 0)
1515             return (ENOMEM);
1516
1517         TRACE_3(TR_FAC_VM, TR_ANON_PROC, "anon proc:%p %lu %u",
1518                 seg, len, 0);
1519     }
1520
1521     newsyd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
1522
1523     newseg->s_ops = &segvn_ops;
1524     newseg->s_data = (void *)newsyd;
1525     newseg->s_szc = seg->s_szc;
1526
1527     newsyd->seg = newseg;
1528     if ((newsyd->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     newsyd->offset = svd->offset;
1534     newsyd->prot = svd->prot;
1535     newsyd->maxprot = svd->maxprot;
1536     newsyd->pageprot = svd->pageprot;
1537     newsyd->type = svd->type;
1538     newsyd->cred = svd->cred;
1539     crhold(newsyd->cred);
1540     newsyd->advice = svd->advice;
1541     newsyd->pageadvice = svd->pageadvice;
1542     newsyd->swresv = svd->swresv;
1543     newsyd->pageswap = svd->pageswap;
1544     newsyd->flags = svd->flags;
1545     newsyd->softlockcnt = 0;
1546     newsyd->softlockcnt_sbase = 0;
1547     newsyd->softlockcnt_send = 0;
1548     newsyd->policy_info = svd->policy_info;
1549     newsyd->rcookie = HAT_INVALID_REGION_COOKIE;
1550
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(newsyd->vp != NULL && amp != NULL);
1559             newsyd->tr_state = SEGVN_TR_INIT;
1560         } else {
1561             newsyd->tr_state = svd->tr_state;
1562         }
1563         newsyd->amp = NULL;
1564         newsyd->anon_index = 0;
1565     } else {
1566         /*
1567          * regions for now are only used on pure vnode segments *
1568          */
1569         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
1570         ASSERT(svd->tr_state == SEGVN_TR_OFF);
1571         newsyd->tr_state = SEGVN_TR_OFF;
1572         if (svd->type == MAP_SHARED) {
1573             newsyd->amp = amp;
1574             ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
1575             amp->refcnt++;
1576             ANON_LOCK_EXIT(&amp->a_rwlock);
1577             newsyd->anon_index = svd->anon_index;
1578         } else {
1579             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      addr += PAGESIZE;
1679      old_idx++;
1680      new_idx++;
1681  } else { /* common case */
1682      if (seg->s_szc != 0) {
1683          /*
1684           * If at least one of anon slots of a
1685           * large page exists then make sure
1686           * all anon slots of a large page
1687           * exist to avoid partial cow sharing
1688           * of a large page in the future.
1689           */
1690          anon_dup_fill_holes(amp->ahp,
1691                              svd->anon_index, newsvd->amp->ahp,
1692                              0, seg->s_size, seg->s_szc,
1693                              svd->vp != NULL);
1694      } else {
1695          anon_dup(amp->ahp, svd->anon_index,
1696                  newsvd->amp->ahp, 0, seg->s_size);
1697      }
1698      hat_clrattr(seg->s_as->a_hat, seg->s_base,
1699                  seg->s_size, PROT_WRITE);
1700  }
1701  /*
1702   */
1703  /*
1704   */
1705  /*
1706   */
1707  /*
1708   * If necessary, create a vpage structure for the new segment.
1709   * Do not copy any page lock indications.
1710   */
1711  if (svd->vpage != NULL) {

```

```

1711     uint_t i;
1712     struct vpage *ovp = svd->vpage;
1713     struct vpage *nvp;
1714
1715     nvp = newsvd->vpage =
1716         kmem_alloc(vpgtob(npages), KM_SLEEP);
1717     for (i = 0; i < npages; i++) {
1718         *nvp = *ovp++;
1719         VPP_CLRPLOCK(nvp++);
1720     }
1721 } else
1722     newsvd->vpage = NULL;
1723
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;
1754
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);
1761
1762     if (!free_pages) {
1763         return;
1764     }
1765
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;
1782
1783     ASSERT(svd->vp != NULL);
1784     ASSERT(cb->hcb_end_addr > cb->hcb_start_addr);
1785     ASSERT(cb->hcb_start_addr >= seg->s_base);
1786
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;
1803
1804     ASSERT(svd->pageswap);
1805     ASSERT(svd->vpage != NULL);
1806
1807     evp = &svd->vpage[seg_page(seg, seg->s_base + seg->s_size)];
1808
1809     for (vp = svd->vpage; vp < evp; vp++) {
1810         if (VPP_ISSWAPRES(vp))
1811             nswappages++;
1812     }
1813
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 nbbase;
1830     size_t nsize;
1831     size_t oswresv;
1832     int reclaim = 1;
1833
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));
1839
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     if (svd->softlockcnt > 0) {
1845         ASSERT(svd->tr_state == SEGVN_TR_OFF);
1846
1847         /*
1848         * If this is shared segment non 0 softlockcnt
1849         * means locked pages are still in use.
1850         */
1851         if (svd->type == MAP_SHARED) {
1852             return (EAGAIN);
1853         }
1854
1855         /*
1856         * since we do have the writers lock nobody can fill
1857         * the cache during the purge. The flush either succeeds
1858         * or we still have pending I/Os.
1859         */
1860         if (reclaim == 1) {
1861             segvn_purge(seg);
1862             reclaim = 0;
1863             goto retry;
1864         }
1865         return (EAGAIN);
1866     }
1867
1868     /*
1869     * Check for bad sizes
1870     */
1871     if (addr < seg->s_base || addr + len > seg->s_base + seg->s_size ||
1872     (len & PAGEOFFSET) || ((uintptr_t)addr & PAGEOFFSET)) {
1873         panic("segvn_unmap");
1874         /*NOTREACHED*/
1875     }
1876
1877     if (seg->s_szc != 0) {
1878         size_t pgsz = page_get_pagesize(seg->s_szc);
1879         int err;
1880         if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(len, pgsz)) {
1881             ASSERT(seg->s_base != addr || seg->s_size != len);
1882             if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
1883                 ASSERT(svd->amp == NULL);
1884                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
1885                 hat_leave_region(seg->s_as->a_hat,
1886                                 svd->rcookie, HAT_REGION_TEXT);
1887                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
1888
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_textunrepr(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         }
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_textunrepr(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 & VVMEBC) &&
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     }

```

```

1975         return (0);
1976     }
1977
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;
1991
1992             ovpage = svd->vpage; /* keep pointer to vpage */
1993
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                 }
2033
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;
2058
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;
2068
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         }
2086
2087         return (0);
2088     }

2089     /*
2090      * Check for end of segment
2091      */
2092     if (addr + len == seg->s_base + seg->s_size) {
2093         if (svd->vpage != NULL) {
2094             size_t nbytes;
2095             struct vpage *ovpage;
2096
2097             ovpage = svd->vpage; /* keep pointer to vpage */
2098
2099             nbytes = vpgtob(npages);
2100             svd->vpage = kmem_alloc(nbytes, KM_SLEEP);
2101             bcopy(ovpage, svd->vpage, nbytes);
2102
2103             /* free up old vpage */
2104             kmem_free(ovpage, vpgtob(opages));
2105         }

```

```

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;
2115
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             }
2126
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             }
2142
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     }
2156
2157     seg->s_size -= len;
2158
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     }
2174     if (svd->pageswap) {
2175         oswresv = svd->swresv;
2176         svd->swresv =
2177             segvn_count_swap_by_vpages(seg);
2178         ASSERT(oswresv >= svd->swresv);
2179         unlen = oswresv - svd->swresv;
2180     } else {
2181         svd->swresv -= len;
2182         ASSERT(svd->swresv == seg->s_size);
2183         unlen = len;
2184     }
2185     anon_unresv_zone(unlen,
2186                     seg->s_as->a_proc->p_zone);
2187     TRACE_3(TR_FAC_VM, TR_ANON_PROC,
2188             "anon proc:%p lu %u", seg, len, 0);
2189 }
2190
2191     return (0);
2192 }
2193
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);
2219
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);
2226
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;
2233
2234     ovpage = svd->vpage; /* keep pointer to vpage */
2235
2236     npages = seg_pages(seg); /* seg has shrunk */
2237     nbytes = vpgtob(npages);
2238     svd->vpage = kmalloc(nbytes, KM_SLEEP);

```

```

2240         bcopy(opage, svd->vpage, nbytes);
2242
2243         npages = seg_pages(nseg);
2244         nbytes = vpgtob(npages);
2245         nsvd->vpage = kmem_alloc(nbytes, KM_SLEEP);
2246
2247         bcopy(&opage[opages - npages], nsvd->vpage, nbytes);
2248
2249         /* free up old vpage */
2250         kmem_free(opage, vpgtob(opages));
2251     }
2252
2253     if (amp == NULL) {
2254         nsvd->amp = NULL;
2255         nsvd->anon_index = 0;
2256     } else {
2257         /*
2258          * Need to create a new anon map for the new segment.
2259          * We'll also allocate a new smaller array for the old
2260          * smaller segment to save space.
2261         */
2262         opages = btop((uintptr_t)(addr - seg->s_base));
2263         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2264         if (amp->refcnt == 1 || svd->type == MAP_PRIVATE) {
2265             /*
2266              * Free up now unused parts of anon_map array.
2267              */
2268             ulong_t an_idx = svd->anon_index + opages;
2269
2270             /*
2271              * Shared anon map is no longer in use. Before
2272              * freeing its pages purge all entries from
2273              * pcache that belong to this amp.
2274             */
2275             if (svd->type == MAP_SHARED) {
2276                 ASSERT(amp->refcnt == 1);
2277                 ASSERT(svd->softlockcnt == 0);
2278                 anonmap_purge(amp);
2279             }
2280
2281             if (amp->a_szc == seg->s_szc) {
2282                 if (seg->s_szc != 0) {
2283                     anon_free_pages(amp->ahp, an_idx, len,
2284                                     seg->s_szc);
2285                 } else {
2286                     anon_free(amp->ahp, an_idx,
2287                               len);
2288                 }
2289             } else {
2290                 ASSERT(svd->type == MAP_SHARED);
2291                 ASSERT(amp->a_szc > seg->s_szc);
2292                 anon_shmap_free_pages(amp, an_idx, len);
2293             }
2294
2295             /*
2296              * Unreserve swap space for the
2297              * unmapped chunk of this segment in
2298              * case it's MAP_SHARED
2299             */
2300             if (svd->type == MAP_SHARED) {
2301                 anon_unresv_zone(len,
2302                                 seg->s_as->a_proc->p_zone);
2303                 amp->swresv -= len;
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;
2313
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;
2347
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 {
2429                 /*
2430                  * Shared anon map is no longer in use. Before
2431                  * freeing its pages purge all entries from
2432                  * pcache that belong to this amp.
2433                  */
2434                 ASSERT(svd->softlockcnt == 0);
2435                 anonmap_purge(amp);
2436             }
2437         }
2438     }
2439 }
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3479
3480
3481
3482
3483
3484
3485
3486
3487
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3579
3580
3581
3582
3583
3584
3585
3586
3587
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3679
3680
3681
3682
3683
3684
3685
3686
3687
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3879
3880
3881
3882
3883
3884
3885
3886
3887
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3979
3980
3981
3982
3983
3984
3985
3986
3987
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4
```

```

2503     svd->pageswap = 0;
2504     svd->cred = NULL;
2505
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);
2514
2515     seg->s_data = NULL;
2516     kmem_cache_free(segvn_cache, svd);
2517 }
2518
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;
2536
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));
2539
2540     if ((amp = svd->amp) != NULL)
2541         anon_index = svd->anon_index + seg_page(seg, addr);
2542
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         }
2566
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     }
2578
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 }
2612
2613 #define PAGE_HANDLED ((page_t *)-1)
2614
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 }
2627
2628 static int stealcow = 1;
2629
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;                  /* 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;
2690
2691     if (svd->flags & MAP_TEXT) {
2692         hat_flag |= HAT_LOAD_TEXT;
2693     }
2694
2695     ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
2696     ASSERT(seg->s_szc == 0);
2697     ASSERT(svd->tr_state != SEGVN_TR_INIT);
2698
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;
2705
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                 }
2721
2722                 prot = VPP_PROT(vpage);
2723                 if ((prot & protchk) == 0)
2724                     return (FC_PROT); /* illegal access type */
2725             } else {
2726                 prot = svd->prot;
2727             }
2728
2729             if (type == F_SOFTLOCK) {
2730                 atomic_inc_ulong((ulong_t *)&svd->softlockcnt);
2731             }
2732
2733             /*
2734             * Always acquire the anon array lock to prevent 2 threads from
2735             * allocating separate anon slots for the same "addr".
2736             */
2737
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             }
2744
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                     }

```

new/usr/src/uts/common/vm/seg\_vn.c

43

```

2767
2768
2769
2770
2771
2772
2773
2774     ASSERT(pp->p_szc == 0);
2775
2776
2777
2778
2779
2780     if (lgrp_optimizations())
2781         page_migrate(seg, addr, &pp, 1);
2782
2783     if (enable_mbit_wa) {
2784         if (rw == S_WRITE)
2785             hat_setmod(pp);
2786         else if (!hat_ismod(pp))
2787             prot &= ~PROT_WRITE;
2788     }
2789
2790     /* If AS_PAGLCK is set in a_flags (via memcntl(2)
2791      * with MC_LOCKAS, MCL_FUTURE) and this is a
2792      * MAP_NORESERVE segment, we may need to
2793      * permanently lock the page as it is being faulted
2794      * for the first time. The following text applies
2795      * only to MAP_NORESERVE segments:
2796
2797     * As per memcntl(2), if this segment was created
2798     * after MCL_FUTURE was applied (a "future"
2799     * segment), its pages must be locked. If this
2800     * segment existed at MCL_FUTURE application (a
2801     * "past" segment), the interface is unclear.
2802
2803     * We decide to lock only if vpage is present:
2804
2805     * - "future" segments will have a vpage array (see
2806     *   as_map), and so will be locked as required
2807
2808     * - "past" segments may not have a vpage array,
2809     *   depending on whether events (such as
2810     *   mprotect) have occurred. Locking if vpage
2811     *   exists will preserve legacy behavior. Not
2812     *   locking if vpage is absent, will not break
2813     *   the interface or legacy behavior. Note that
2814     *   allocating vpage here if it's absent requires
2815     *   upgrading the segvn reader lock, the cost of
2816     *   which does not seem worthwhile.
2817
2818     * Usually testing and setting VPP_ISPPLLOCK and
2819     * VPP_SETPLOCK requires holding the segvn lock as
2820     * writer, but in this case all readers are
2821     * serializing on the anon array lock.
2822
2823     if (AS_ISPGLCK(seg->s_as) && vpage != NULL &&
2824         (svd->flags & MAP_NORESERVE) &&
2825         !VPP_ISPPLLOCK(vpage)) {
2826         proc_t *p = seg->s_as->a_proc;
2827         ASSERT(svd->type == MAP_PRIVATE);
2828         mutex_enter(&p->p_lock);
2829         if (rctl_incr_locked_mem(p, NULL, PAGESIZE,
2830             1) == 0) {
2831             claim = VPP_PROT(vpage) & PROT_WRITE;
2832             if (page_pp_lock(pp, claim, 0)) {
2833                 VPP_SETPLOCK(vpage);

```

[new/usr/src/uts/common/vm/seg\\_vn.c](#)

```

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;
2863
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(opp != 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(opp == 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     /*
2981      * We are low on memory, the page is not private, page is not large,
2982      * not shared, not modified, not 'locked' or if we have it 'locked'
2983      * (i.e., p_cowcnt == 1 and p_lckcnt == 0, which also implies
2984      * that the page is not shared) and if it doesn't have any
2985      * translations. page_struct_lock isn't needed to look at p_cowcnt
2986      * and p_lckcnt because we first get exclusive lock on page.
2987     */
2988     (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         }
3003         /*
3004          * hat_unload() might sync back someone else's recent
3005          * modification, so check again.
3006         */
3007         if (!hat_ismod(opp) && !hat_page_is_mapped(opp))
3008             pageflags |= STEAL_PAGE;
3009     }
3010     /*
3011      * If we have a vpage pointer, see if it indicates that we have
3012      * ``locked'' the page we map -- if so, tell anon_private to
3013      * transfer the locking resource to the new page.
3014      *
3015      * See Statement at the beginning of segvn_lockop regarding
3016      * the way lockcnts/cowcnts are handled during COW.
3017      *
3018      */
3019     if (vpage != NULL && VPP_ISPPLOCK(vpage))
3020         pageflags |= LOCK_PAGE;
3021     /*
3022      * Allocate a private page and perform the copy.
3023      * For MAP_NORESERVE reserve swap space now, unless this
3024      * is a cow fault on an existing anon page in which case
3025      * MAP_NORESERVE will have made advance reservations.
3026      */
3027     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 npggs;

3113     VM_STAT_ADD(segvnvmstats.relocatepages[0]);

3115     ASSERT(repl_szc != 0);
3116     npggs = 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 pfns;
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         pfns = page_pptonum(pp);
3214         if (i == 0) {
3215             if (!IS_P2ALIGNED(pfn, totnpgs)) {
3216                 contig = 0;
3217             } else {
3218                 first_pfn = pfns;
3219             }
3220         } else if (contig && pfns != 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 (pfns - 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(segvnmstats.fullszcpages[9]);
3303     return (1);
3304 }

3305 /*
3306  * Create physically contiguous pages for [vp, off] - [vp, off +
3307  * page_size(szc)) range and for private segment return them in ppa array.
3308  * Pages are created either via IO or relocations.
3309  *
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 */

3317 static int
3318 segvn_fill_vp_pages(struct segvn_data *svd, vnode_t *vp, u_offset_t off,
3319     uint_t szc, page_t **ppa, page_t **ppplist, uint_t *ret_pszc,
3320     int *downsize)
3321 {
3322     page_t *pplist = *ppplist;
3323     size_t pgsz = page_get_pagesize(szc);
3324     pgcnt_t pages = btop(pgsz);
3325     ulong_t start_off = off;
3326     u_offset_t eoff = off + pgSz;
3327     spgcnt_t nreloc;
3328     u_offset_t io_off = off;
3329     size_t io_len;
3330     page_t *io_pplist = NULL;
3331     page_t *done_pplist = NULL;
3332     pgcnt_t pgidx = 0;
3333     page_t *pp;
3334     page_t *newpp;
3335     page_t *targpp;
3336     page_t *targpp;
3337     page_t *targpp;
3338     int io_err = 0;
3339     int i;
3340     pfn_t pfns;
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;

3348     VM_STAT_ADD(segvnmstats.fill_vp_pages[0]);

3349     ASSERT(szc != 0);
3350     ASSERT(pplist->p_szc == szc);

3351     /*
3352      * downsize will be set to 1 only if we fail to lock pages. this will
3353      * allow subsequent faults to try to relocate the page again. If we
3354      * fail due to misalignment don't downsize and let the caller map the
3355      * whole region with small mappings to avoid more faults into the area
3356      * where we can't get large pages anyway.
3357      */

```

```

3361     *downsize = 0;

3362     while (off < eoff) {
3363         newpp = plist;
3364         ASSERT(newpp != NULL);
3365         ASSERT(PAGE_EXCL(newpp));
3366         ASSERT(!PP_ISFREE(newpp));
3367         /*
3368          * we pass NULL for nrelop to page_lookup_create()
3369          * so that it doesn't relocate. We relocate here
3370          * later only after we make sure we can lock all
3371          * pages in the range we handle and they are all
3372          * aligned.
3373         */
3374         pp = page_lookup_create(vp, off, SE_SHARED, newpp, NULL, 0);
3375         ASSERT(pp != NULL);
3376         ASSERT(!PP_ISFREE(pp));
3377         ASSERT(pp->p_vnode == vp);
3378         ASSERT(pp->p_offset == off);
3379         if (pp == newpp) {
3380             VM_STAT_ADD(segvnmstats.fill_vp_pages[1]);
3381             page_sub(&plist, pp);
3382             ASSERT(PAGE_EXCL(pp));
3383             ASSERT(page_iolock_assert(pp));
3384             page_list_concat(&io_pplist, &pp);
3385             off += PAGESIZE;
3386             continue;
3387         }
3388     }
3389     VM_STAT_ADD(segvnmstats.fill_vp_pages[2]);
3390     pfns = page_pptonum(pp);
3391     pszc = pp->p_szc;
3392     if (pszc >= szc && targ_pplist == NULL && io_pplist == NULL &&
3393         IS_P2ALIGNED(pfns, 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         *pplist = NULL;
3401         VM_STAT_ADD(segvnmstats.fill_vp_pages[3]);
3402         return (1);
3403     }
3404     if (pszc >= szc) {
3405         page_unlock(pp);
3406         segvn_faultvnmppss_align_err1++;
3407         goto out;
3408     }
3409     ppages = page_get_pagecnt(pszc);
3410     if (!IS_P2ALIGNED(pfns, ppages)) {
3411         ASSERT(pszc > 0);
3412         /*
3413          * sizing down to pszc won't help.
3414          */
3415         page_unlock(pp);
3416         segvn_faultvnmppss_align_err2++;
3417         goto out;
3418     }
3419     pfns = page_pptonum(newpp);
3420     if (!IS_P2ALIGNED(pfns, ppages)) {
3421         ASSERT(pszc > 0);
3422         /*
3423          * sizing down to pszc won't help.
3424          */
3425         page_unlock(pp);
3426         segvn_faultvnmppss_align_err3++;
3427     }

```

```

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 <= pszcz);
3489     if (pszcz != 0 & !group_page_trylock(pp, SE_EXCL)) {
3490         VM_STAT_ADD(segvnvmstats.fill_vp_pages[10]);
3491         page_unlock(pp);
3492         *downsize = 1;
3493     }

```

```

3493         *ret_pszc = pp->p_szc;
3494         goto out;
3495     }
3496     VM_STAT_ADD(segvnvmstats.fill_vp_pages[11]);
3497     /*
3498      * page szc should have changed before the entire group was
3499      * locked. reread page szc.
3500     */
3501     pszcz = pp->p_szc;
3502     ppages = page_get_pagecnt(pszcz);
3503
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 = plist;
3511         page_sub(&plist, 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(segvnmstats.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     }
3595
3596     if (svd->type == MAP_PRIVATE) {
3597         VM_STAT_ADD(segvnmstats.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(segvnmstats.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 plist.
3631     */
3632     if (io_pplist != NULL) {
3633         VM_STAT_ADD(segvnmstats.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, &plist);
3644         plist = io_pplist;
3645     }
3646     tmp_pplist = NULL;
3647     while (targ_pplist != NULL) {
3648         VM_STAT_ADD(segvnmstats.fill_vp_pages[21]);
3649         pp = targ_pplist;
3650         ASSERT(PAGE_EXCL(pp));
3651         page_sub(&targ_pplist, pp);
3652
3653         pszc = pp->p_szc;
3654         ppages = page_get_pagecnt(pszc);
3655         ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));
3656
3657         if (pszc != 0) {
3658             group_page_unlock(pp);
3659         }
3660         page_unlock(pp);
3661
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);
3667
3668         ASSERT(IS_P2ALIGNED(page_pptonum(pp), ppages));
3669
3670         /* relink replacement page */
3671         page_list_concat(&tmp_pplist, &pp);
3672         while (--ppages != 0) {
3673             VM_STAT_ADD(segvnmstats.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(segvnmstats.fill_vp_pages[23]);
3682         page_list_concat(&tmp_pplist, &plist);
3683         plist = tmp_pplist;
3684     }
3685
3686     /*
3687      * at this point all pages are either on done_pplist or
3688      * plist. They can't be all on done_pplist otherwise
3689      * we'd've been done.
3690     */
3691     ASSERT(plist != NULL);

```

```

3691     if (nios != 0) {
3692         VM_STAT_ADD(segvmstats.fill_vp_pages[24]);
3693         pp = plist;
3694         do {
3695             VM_STAT_ADD(segvmstats.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) != plist);
3701
3702         pp = done_plist;
3703         do {
3704             VM_STAT_ADD(segvmstats.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_plist);
3710
3711         while (plist != NULL) {
3712             VM_STAT_ADD(segvmstats.fill_vp_pages[27]);
3713             pp = plist;
3714             page_sub(&done_plist, pp);
3715             page_free(pp, 0);
3716         }
3717
3718         while (done_plist != NULL) {
3719             VM_STAT_ADD(segvmstats.fill_vp_pages[28]);
3720             pp = done_plist;
3721             page_sub(&done_plist, pp);
3722             page_unlock(pp);
3723         }
3724         *plist = NULL;
3725         return (0);
3726     }
3727     ASSERT(plist == *plist);
3728     if (io_err) {
3729         VM_STAT_ADD(segvmstats.fill_vp_pages[29]);
3730         /*
3731          * don't downsize on io error.
3732          * see if vop_getpage succeeds.
3733          * plist may still be used in this case
3734          * for relocations.
3735          */
3736         return (0);
3737     }
3738     VM_STAT_ADD(segvmstats.fill_vp_pages[30]);
3739     page_free_replacement_page(plist);
3740     page_create_putback(pages);
3741     *plist = 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
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4498
4499
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4598
4599
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4698
4699
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4789
4790
4791
4792
4793
4794
4795
4796
4797
4797
4798
4798
4799
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4878
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4898
4899
4899
4900
4901
4902
4903
4904
4905
4906
4907
4908
4909
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4978
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4998
4999
4999
5000
5001
5002
5003
5004
5005
5006
5007
5008
5009
5009
5010
5011
5012
5013
5014
5015
5016
5017
5018
5019
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
5049
5050
5051
5052
5053
5054
5055
5056
5057
5058
5059
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5079
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5098
5099
5099
5100
5101
5102
5103
5104
5105
5106
5107
5108
5109
5109
5110
5111
5112
5113
5114
5115
5116
5117
5118
5119
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
5149
5150
5151
5152
5153
5154
5155
5156
5157
5158
5159
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5198
5199
5199
5200
5201
5202
5203
5204
5205
5206
5207
5208
5209
5209
5210
5211
5212
5213
5214
5215
5216
5217
5218
5219
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
5249
5250
5251
5252
5253
5254
5255
5256
5257
5258
5259
5259
5260
5261
5262
5263
5264
5265
5266
5267
5268
5269
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5298
5299
5299
5300
5301
5302
5303
5304
5305
5306
5307
5308
5309
5309
5310
5311
5312
5313
5314
5315
5316
5317
5318
5319
5319
5320
5321
5322
5323
5324
5325
5326
5327
5328
5329
5329
5330
5331
5332
5333
5334
5335
5336
5337
5338
5339
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
5349
5350
5351
5352
5353
5354
5355
5356
5357
5358
5359
5359
5360
5361
5362
5363
5364
5365
5366
5367
5368
5369
5369
5370
5371
5372
5373
5374
5375
5376
5377
5378
5379
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
5399
5400
5401
5402
5403
5404
5405
5406
5407
5408
5409
5409
5410
5411
5412
5413
5414
5415
5416
5417
5418
5419
5419
5420
5421
5422
5423
5424
5425
5426
5427
5428
5429
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
5449
5450
5451
5452
5453
5454
5455
5456
5457
5458
5459
5459
5460
5461
5462
5463
5464
5465
5466
5467
5468
5469
5469
5470
5471
5472
5473
5474
5475
5476
5477
5478
5479
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5498
5499
5499
5500
5501
5502
5503
5504
5505
5506
5507
5508
5509
5509
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
5549
5550
5551
5552
5553
5554
5555
5556
5557
5558
5559
5559
5560
5561
5562
5563
5564
5565
5566
5567
5568
5569
5569
5570
5571
5572
5573
5574
5575
5576
5577
5578
5579
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5598
5599
5599
5600
5601
5602
5603
5604
5605
5606
5607
5608
5609
5609
5610
5611
5612
5613
5614
5615
5616
5617
5618
5619
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
5649
5650
5651
5652
5653
5654
5655
5656
5657
5658
5659
5659
5660
5661
5662
5663
5664
5665
5666
5667
5668
5669
5669
5670
5671
5672
5673
5674
5675
5676
5677
5678
5679
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5689
5690
5691
5692
5
```

```

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, segvnmstats.fltvnpages[0]);
3844     VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnmstats.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         } else {
3866             prot = svd->prot;
3867             /* caller has already done segment level protection check. */
3868         }
3869     }

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

3875     if (rw == S_WRITE && segtype == MAP_PRIVATE) {
3876         SEGVN_VMSSTAT_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, segvnmstats.fltvnpages[3]);

3886     for (;;) {
3887         adjszc_chk = 0;
3888         for (; a < lpgeaddr; a += pqsz, off += pqsz, 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     }
3908
3909 again:   if (IS_P2ALIGNED(a, maxpgsz) && a != NULL) {
3910     ASSERT(IS_P2ALIGNED(aindx, maxpages));
3911     ANON_LOCK_ENTER(&a_rwlock, RW_READER);
3912     anon_array_enter(&am, aindx, &an_cookie);
3913     if (anon_get_ptr(am->ahp, aindx) == NULL) {
3914         SEGVN_VMSTAT_FLTVNPAGES(5);
3915         ASSERT(anon_pages(am->ahp, aindx,
3916                           maxpages) == maxpages);
3917         anon_array_exit(&an_cookie);
3918         ANON_LOCK_EXIT(&a_rwlock);
3919         err = segvn_fault_anonpages(hat, seg,
3920                                     a, a + maxpgsz, type, rw,
3921                                     MAX(a, addr),
3922                                     MIN(a + maxpgsz, eaddr), brkcow);
3923     }
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(am->ahp, aindx,
3937                       maxpages) == 0);
3938     SEGVN_VMSTAT_FLTVNPAGES(7);
3939     anon_array_exit(&an_cookie);
3940     ANON_LOCK_EXIT(&a_rwlock);
3941 }
3942
3943 ASSERT(!brkcow || IS_P2ALIGNED(a, maxpgsz));
3944 ASSERT(!tron || IS_P2ALIGNED(a, maxpgsz));
3945
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     }

```

new/usr/src/uts/common/vm/seg\_vn.c

61

```

3955
3956         }
3957         if (type == F_SOFTLOCK) {
3958             atomic_add_long((ulong_t *)&svd->softlockcnt,
3959             pages);
3960     }
3961
3962     pplist = NULL;
3963     physcontig = 0;
3964     ppa[0] = NULL;
3965     if (!brkcow && !tron && szc &&
3966         !page_exists_physcontig(vp, off, szc,
3967         segtype == MAP_PRIVATE ? ppa : NULL)) {
3968         SEGVN_VMSTAT_FLTVNPAGES(9);
3969         if (page_alloc_pages(vp, seg, a, &pplist, NULL,
3970             szc, 0, 0) && type != F_SOFTLOCK) {
3971             SEGVN_VMSTAT_FLTVNPAGES(10);
3972             pszc = 0;
3973             ierr = -1;
3974             alloc_failed |= (1 << szc);
3975             break;
3976         }
3977         if (pplist != NULL &&
3978             vp->v_mpssdata == SEGVN_PAGEIO) {
3979             int downsize;
3980             SEGVN_VMSTAT_FLTVNPAGES(11);
3981             physcontig = segvn_fill_vp_pages(svd,
3982                 vp, off, szc, ppa, &pplist,
3983                 &pszc, &downsize);
3984             ASSERT(!physcontig || pplist == NULL);
3985             if (!physcontig && downsize &&
3986                 type != F_SOFTLOCK) {
3987                 ASSERT(pplist == NULL);
3988                 SEGVN_VMSTAT_FLTVNPAGES(12);
3989                 ierr = -1;
3990                 break;
3991             }
3992             ASSERT(!physcontig ||
3993                 segtype == MAP_PRIVATE ||
3994                 ppa[0] == NULL);
3995             if (physcontig && ppa[0] == NULL) {
3996                 physcontig = 0;
3997             }
3998         } else if (!brkcow && !tron && szc && ppa[0] != NULL) {
3999             SEGVN_VMSTAT_FLTVNPAGES(13);
4000             ASSERT(segtype == MAP_PRIVATE);
4001             physcontig = 1;
4002         }
4003
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
4012         if (ierr == 0) {
4013             for (i = 0; i < pages; i++) {
4014                 ASSERT(PAGE_LOCKED(ppa[i]));
4015                 ASSERT(!PP_ISFREE(ppa[i]));
4016                 ASSERT(ppa[i]->p_vnode == vp);
4017                 ASSERT(ppa[i]->p_offset ==
4018                     off + (i << PAGESHIFT));
4019             }
4020 #endif /* DEBUG */

```

new/usr/src/uts/common/vm/seg\_vn.c

```

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

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

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

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

```

```

4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121 #ifdef DEBUG
4122
4123
4124
4125
4126 #endif /* DEBUG */

4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4147
4148
4149
4150
4151
4152
        page_unlock(ppa[i]);
    }
    anon_array_exit(&an_cookie);
    ANON_LOCK_EXIT(&a_rwlock);
    if (pplist != NULL) {
        page_free_replacement_page(pplist);
        page_create_putback(pages);
    }
    SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
    if (szc < seg->s_szc) {
        SEGVN_VMSTAT_FLTVNPAGES(26);
        /*
         * For private segments SOFTLOCK
         * either always breaks cow (any rw
         * type except S_READ_NOCOW) or
         * address space is locked as writer
         * (S_READ_NOCOW case) and anon slots
         * can't show up on second check.
         * Therefore if we are here for
         * SOFTLOCK case it must be a cow
         * break but cow break never reduces
         * szc. text replication (tron) in
         * this case works as cow break.
         * Thus the assert below.
         */
        ASSERT(!brkcw && !tron &&
               type != F_SOFTLOCK);
        pszc = seg->s_szc;
        ierr = -2;
        break;
    }
    ASSERT(IS_P2ALIGNED(a, maxpgsz));
    goto again;
}

if (amp != NULL) {
    ulong_t taindex = P2ALIGN(aindex, maxpages);
    ASSERT(!anon_pages(amp->ahp, taindex, maxpages));
}

if (brkcw || tron) {
    ASSERT(amp != NULL);
    ASSERT(pplist == NULL);
    ASSERT(szc == seg->s_szc);
    ASSERT(IS_P2ALIGNED(a, maxpgsz));
    ASSERT(IS_P2ALIGNED(aindex, maxpages));
    SEGVN_VMSTAT_FLTVNPAGES(27);
    ierr = anon_map_privatepages(amp, aindex, szc,
                                seg, a, prot, ppa, vpage, segvn_anypgsz,
                                tron ? PG_LOCAL : 0, svd->cred);
    if (ierr != 0) {
        SEGVN_VMSTAT_FLTVNPAGES(28);
        anon_array_exit(&an_cookie);
        ANON_LOCK_EXIT(&a_rwlock);
        SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
        err = FC_MAKE_ERR(ierr);
        goto out;
    }
    ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));
    /*
     * p_szc can't be changed for locked
     * swapfs pages.
     */
    ASSERT(svd->rcookie ==

```

```

4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201 #ifdef DEBUG
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212 #endif /* DEBUG */
4213
4214
4215
4216
4217
4218
        HAT_INVALID_REGION_COOKIE);
    hat_memload_array(hat, a, pgsz, ppa, prot,
                      hat_flag);
    if (!(hat_flag & HAT_LOAD_LOCK)) {
        SEGVN_VMSTAT_FLTVNPAGES(29);
        for (i = 0; i < pages; i++) {
            page_unlock(ppa[i]);
        }
    }
    anon_array_exit(&an_cookie);
    ANON_LOCK_EXIT(&a_rwlock);
    goto next;
}

ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
       (!svd->pageprot && svd->prot == (prot & vpprot)));
pfn = page_pptonum(ppa[0]);
/*
 * hat_page_demote() needs an SE_EXCL lock on one of
 * constituent page_t's and it decreases root's p_szc
 * last. This means if root's p_szc is equal szc and
 * all its constituent pages are locked
 * hat_page_demote() that could have changed p_szc to
 * szc is already done and no new have page_demote()
 * can start for this large page.
 */
/*
 * we need to make sure same mapping size is used for
 * the same address range if there's a possibility the
 * address is already mapped because hat layer panics
 * when translation is loaded for the range already
 * mapped with a different page size. We achieve it
 * by always using largest page size possible subject
 * to the constraints of page size, segment page size
 * and page alignment. Since mappings are invalidated
 * when those constraints change and make it
 * impossible to use previously used mapping size no
 * mapping size conflicts should happen.
*/
chkszc:
if ((pszc = ppa[0]->p_szc) == szc &&
    IS_P2ALIGNED(pfn, pages)) {
    SEGVN_VMSTAT_FLTVNPAGES(30);
    for (i = 0; i < pages; i++) {
        ASSERT(PAGE_LOCKED(ppa[i]));
        ASSERT(!PP_ISFREE(ppa[i]));
        ASSERT(page_pptonum(ppa[i]) ==
               pfn + i);
        ASSERT(ppa[i]->p_szc == szc);
        ASSERT(ppa[i]->p_vnode == vp);
        ASSERT(ppa[i]->p_offset ==
               off + (i << PAGESHIFT));
    }
}
/*
 * All pages are of szc we need and they are
 * all locked so they can't change szc. load
 * translations.
 *
 * if page got promoted since last check

```

```

4219             * we don't need plist.
4220             /*
4221             if (plist != NULL) {
4222                 page_free_replacement_page(plist);
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             }
4249             if (!(hat_flag & HAT_LOAD_LOCK)) {
4250                 for (i = 0; i < pages; i++) {
4251                     page_unlock(ppa[i]);
4252                 }
4253             if (amp != NULL) {
4254                 anon_array_exit(&an_cookie);
4255                 ANON_LOCK_EXIT(&a_rwlock);
4256             }
4257             goto next;
4258         }
4259     }
4260     /*
4261     * See if upsize is possible.
4262     */
4263     if (pszc > szc && szc < seg->s_szc &&
4264         (segvn_anypgsz_vnode || pszc >= seg->s_szc)) {
4265         pgcnt_t aphase;
4266         uint_t pszc1 = MIN(pszc, seg->s_szc);
4267         ppgsz = page_get_pagesize(pszc1);
4268         ppages = btop(ppgsz);
4269         aphase = btop(P2PHASE((uintptr_t)a, ppgsz));
4270
4271         ASSERT(type != F_SOFTLOCK);
4272
4273         SEGVN_VMSTAT_FLTVNPAGES(31);
4274         if (aphase != P2PHASE(pfn, ppages)) {
4275             segvn_faultvnmppss_align_err4++;
4276         } else {
4277             SEGVN_VMSTAT_FLTVNPAGES(32);
4278             if (plist != NULL) {
4279                 page_t *pl = plist;
4280                 page_free_replacement_page(pl);
4281                 page_create_putback(pages);
4282             }
4283         }
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(&a_rwlock);
4291             }
4292             pszc = pszc1;
4293             ierr = -2;
4294             break;
4295         }
4296     }
4297
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))) {
4308
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);
4315
4316             SEGVN_VMSTAT_FLTVNPAGES(33);
4317
4318             if (plist != NULL) {
4319                 page_t *pl = plist;
4320                 page_free_replacement_page(pl);
4321                 page_create_putback(pages);
4322             }
4323
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(&a_rwlock);
4330             }
4331             ierr = -1;
4332             break;
4333         }
4334         if (szc != 0 && !xhat && !upgrdfail) {
4335             segvn_faultvnmppss_align_err5++;
4336         }
4337         SEGVN_VMSTAT_FLTVNPAGES(34);
4338         if (plist != NULL) {
4339             page_free_replacement_page(plist);
4340             page_create_putback(pages);
4341         }
4342         SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4343             prot, vpprot);
4344         if (upgrdfail && segvn_anypgsz_vnode) {
4345             /*
4346             * SOFTLOCK case */
4347             hat_memload_array_region(hat, a, pgsz,
4348                 ppa, prot & vpprot, hat_flag,
4349                 svd->rcookie);
4350         } else {
4351             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 }
4368
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 }
4377
4378 if (pszc > szc) {
4379     kmutex_t *szcmctx;
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     szcmctx = page_szc_lock(ppa[0]);
4390     pszcz = ppa[0]->p_szc;
4391     ASSERT(szcmctx != NULL || pszcz == 0);
4392     ASSERT(ppa[0]->p_szc <= pszcz);
4393     if (pszcz <= szc) {
4394         SEGVN_VMSTAT_FLTVNPAGES(36);
4395         if (szcmctx != NULL) {
4396             mutex_exit(szcmctx);
4397         }
4398         goto chkszc;
4399     }
4400     if (pplist != NULL) {
4401         /*
4402          * page got promoted since last check.
4403          * we don't need preallocated 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(szcmctx);
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 }
4426
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 }
4447
4448 SEGVN_VMSTAT_FLTVNPAGES(39);
4449
4450 if (pplist != NULL) {
4451     segvn_relocate_pages(ppa, plist);
4452 } else {
4453     ASSERT(type == F_SOFTLOCK);
4454     SEGVN_VMSTAT_FLTVNPAGES(40);
4455 }
4456
4457 SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot);
4458
4459 if (pplist == NULL && segvn_anypgsz_vnode == 0) {
4460     ASSERT(type == F_SOFTLOCK);
4461     for (i = 0; i < pages; i++) {
4462         ASSERT(ppa[i]->p_szc < szc);
4463         hat_memload_region(hat,
4464                           a + (i << PAGESHIFT),
4465                           ppa[i], prot & vpprot, hat_flag,
4466                           svd->rcookie);
4467     }
4468 } else {
4469     ASSERT(pplist != NULL || type == F_SOFTLOCK);
4470     hat_memload_array_region(hat, a, pgsz, ppa,
4471                             prot & vpprot, hat_flag, svd->rcookie);
4472 }
4473
4474 if (!(hat_flag & HAT_LOAD_LOCK)) {
4475     for (i = 0; i < pages; i++) {
4476         ASSERT(PAGE_SHARED(ppa[i]));
4477         page_unlock(ppa[i]);
4478     }
4479 }
4480
4481 if (amp != NULL) {
4482     anon_array_exit(&an_cookie);
4483 }
```

```

4483             ANON_LOCK_EXIT(&_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 to 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 = bttop(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     out:
4568     kmem_free(ppa, ppasize);
4569     if (!err && !vop_size_err) {
4570         SEGVN_VMSTAT_FLTVNPAGES(45);
4571         return (0);
4572     }
4573     if (type == F_SOFTLOCK && a > lpgaddr) {
4574         SEGVN_VMSTAT_FLTVNPAGES(46);
4575         segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4576     }
4577     if (!vop_size_err) {
4578         SEGVN_VMSTAT_FLTVNPAGES(47);
4579         return (err);
4580     }
4581     ASSERT(brkcow || tron || type == F_SOFTLOCK);
4582     /*
4583      * Large page end is mapped beyond the end of file and it's a cow
4584      * fault (can be a text replication induced cow) or softlock so we can't
4585      * reduce the map area. For now just demote the segment. This should
4586      * really only happen if the end of the file changed after the mapping
4587      * was established since when large page segments are created we make
4588      * sure they don't extend beyond the end of the file.
4589      */
4590     SEGVN_VMSTAT_FLTVNPAGES(48);
4591
4592     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4593     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4594     err = 0;
4595     if (seg->s_szc != 0) {
4596         segvn_fltvnpages_clrszc_cnt++;
4597         ASSERT(svd->softlockcnt == 0);
4598         err = segvn_clrszc(seg);
4599         if (err != 0) {
4600             segvn_fltvnpages_clrszc_err++;
4601         }
4602     }
4603     ASSERT(err || seg->s_szc == 0);
4604     SEGVN_LOCK_DOWNGRADE(seg->s_as, &svd->lock);
4605     /* segvn_fault will do its job as if szc had been zero to begin with */
4606     return (err == 0 ? IE_RETRY : FC_MAKE_ERR(err));
4607
4608 }
4609 /*
4610  * This routine will attempt to fault in one large page.
4611  * it will use smaller pages if that fails.
4612  * It should only be called for pure anonymous segments.
4613  */

```

```

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 seftype = 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;
4642
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);
4651
4652     ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
4653
4654     VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnmstats.fltnanpages[0]);
4655     VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnmstats.fltnanpages[1]);
4656
4657     if (svd->flags & MAP_TEXT) {
4658         hat_flag |= HAT_LOAD_TEXT;
4659     }
4660
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(segvnmstats.fltnanpages[2]);
4678 } else {
4679     prot = svd->prot;
4680     /* caller has already done segment level protection check. */
4681 }
```

```

4681 }
4682
4683 ppa = kmem_cache_alloc(segvn_szc_cache[ppaszc], KM_SLEEP);
4684 ANON_LOCK_ENTER(&ppa->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(segvnmstats.fltnanpages[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)aaddr,
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(segvnmstats.fltnanpages[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(segvnmstats.fltnanpages[6]);
4727             err = FC_MAKE_ERR(ierr);
4728             goto error;
4729         }
4730     }
4731
4732     ASSERT(!IS_VMODSORT(ppa[0]->p vnode));
4733
4734     ASSERT(segtype == MAP_SHARED ||
4735            ppa[0]->p_szc <= szc);
4736     ASSERT(segtype == MAP_PRIVATE ||
4737            ppa[0]->p_szc >= szc);
4738
4739     /*
4740      * Handle pages that have been marked for migration
4741      */
4742     if (lgrp_optimizations())
4743         page_migrate(seg, a, ppa, pages);
4744
4745     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
4746 }
```

```

4748
4749
4750
4752
4753     if (segtype == MAP_SHARED) {
4754         vpprot |= PROT_WRITE;
4755
4756         hat_memload_array(hat, a, pgsz, ppa,
4757                         prot & vpprot, hat_flag);
4758
4759         if (hat_flag & HAT_LOAD_LOCK) {
4760             VM_STAT_ADD(segvmstats.fltnpages[7]);
4761         } else {
4762             VM_STAT_ADD(segvmstats.fltnpages[8]);
4763             for (i = 0; i < pages; i++)
4764                 page_unlock(ppa[i]);
4765         }
4766         if (vpage != NULL)
4767             vpage += pages;
4768
4769         anon_array_exit(&cookie);
4770         adjszc_chk = 1;
4771     }
4772     if (a == lpgaddr)
4773         break;
4774     ASSERT(a < lpgaddr);
4775     /*
4776      * ierr == -1 means we failed to allocate a large page.
4777      * so do a size down operation.
4778      *
4779      * ierr == -2 means some other process that privately shares
4780      * pages with this process has allocated a larger page and we
4781      * need to retry with larger pages. So do a size up
4782      * operation. This relies on the fact that large pages are
4783      * never partially shared i.e. if we share any constituent
4784      * page of a large page with another process we must share the
4785      * entire large page. Note this cannot happen for SOFTLOCK
4786      * case, unless current address (a) is at the beginning of the
4787      * next page size boundary because the other process couldn't
4788      * have relocated locked pages.
4789      */
4790     ASSERT(ierr == -1 || ierr == -2);
4791
4792     if (segvn_anypgsz) {
4793         ASSERT(ierr == -2 || szc != 0);
4794         ASSERT(ierr == -1 || szc < seg->s_szc);
4795         szc = (ierr == -1) ? szc - 1 : szc + 1;
4796     } else {
4797         /*
4798          * For non COW faults and segvn_anypgsz == 0
4799          * we need to be careful not to loop forever
4800          * if existing page is found with szc other
4801          * than 0 or seg->s_szc. This could be due
4802          * to page relocations on behalf of DR or
4803          * more likely large page creation. For this
4804          * case simply re-size to existing page's szc
4805          * if returned by anon_map_getpages().
4806          */
4807         if (ppa_szc == (uint_t)-1) {
4808             szc = (ierr == -1) ? 0 : seg->s_szc;
4809         } else {
4810             ASSERT(ppa_szc <= seg->s_szc);
4811             ASSERT(ierr == -2 || ppa_szc < szc);
4812             ASSERT(ierr == -1 || ppa_szc > szc);
4813             szc = ppa_szc;
4814         }
4815     }

```

```

4816
4817     pgsz = page_get_pagesize(szc);
4818     pages = btop(pgsz);
4819     ASSERT(type != F_SOFTLOCK || ierr == -1 ||
4820            (IS_P2ALIGNED(a, pgsz) && IS_P2ALIGNED(lpgaddr, pgsz)));
4821     if (type == F_SOFTLOCK) {
4822         /*
4823          * For softlocks we cannot reduce the fault area
4824          * (calculated based on the largest page size for this
4825          * segment) for size down and a is already next
4826          * page size aligned as asserted above for size
4827          * ups. Therefore just continue in case of softlock.
4828          */
4829     VM_STAT_ADD(segvmstats.fltnpages[9]);
4830     continue; /* keep lint happy */
4831 } else if (ierr == -2) {
4832     /*
4833      * Size up case. Note lpgaddr may only be needed for
4834      * softlock case so we don't adjust it here.
4835      */
4836     VM_STAT_ADD(segvmstats.fltnpages[10]);
4837     a = (caddr_t)P2ALIGN(uintptr_t)a, pgsz);
4838     ASSERT(a >= lpgaddr);
4839     lpgaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4840     aindx = svd->anon_index + seg_page(seg, a);
4841     vpage = (svd->vpage != NULL) ?
4842             &svd->vpage[seg_page(seg, a)] : NULL;
4843 } else {
4844     /*
4845      * Size down case. Note lpgaddr may only be needed for
4846      * softlock case so we don't adjust it here.
4847      */
4848     VM_STAT_ADD(segvmstats.fltnpages[11]);
4849     ASSERT(IS_P2ALIGNED(a, pgsz));
4850     ASSERT(IS_P2ALIGNED(lpgaddr, pgsz));
4851     lpgaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4852     ASSERT(a < lpgaddr);
4853     if (a < addr) {
4854         /*
4855          * The beginning of the large page region can
4856          * be pulled to the right to make a smaller
4857          * region. We haven't yet faulted a single
4858          * page.
4859          */
4860     VM_STAT_ADD(segvmstats.fltnpages[12]);
4861     a = (caddr_t)P2ALIGN(uintptr_t)addr, pgsz);
4862     ASSERT(a >= lpgaddr);
4863     aindx = svd->anon_index + seg_page(seg, a);
4864     vpage = (svd->vpage != NULL) ?
4865             &svd->vpage[seg_page(seg, a)] : NULL;
4866 } }
4867 VM_STAT_ADD(segvmstats.fltnpages[13]);
4868 ANON_LOCK_EXIT(&a_rwlock);
4869 kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
4870 return (0);
4871 error:
4872 VM_STAT_ADD(segvmstats.fltnpages[14]);
4873 ANON_LOCK_EXIT(&a_rwlock);
4874 kmem_cache_free(segvn_szc_cache[ppaszc], ppa);
4875 if (type == F_SOFTLOCK && a > lpgaddr) {
4876     VM_STAT_ADD(segvmstats.fltnpages[15]);
4877     segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4878 }
4879 return (err);

```

```

4879 }
4880 int fltadvice = 1; /* set to free behind pages for sequential access */
4881 /*
4882 * This routine is called via a machine specific fault handling routine.
4883 * It is also called by software routines wishing to lock or unlock
4884 * a range of addresses.
4885 *
4886 * Here is the basic algorithm:
4887 *   If unlocking
4888 *     Call segvn_softunlock
4889 *   Return
4890 *
4891 * Checking and set up work
4892 *   If we will need some non-anonymous pages
4893 *     Call VOP_GETPAGE over the range of non-anonymous pages
4894 *   endif
4895 *   Loop over all addresses requested
4896 *     Call segvn_faultpage passing in page list
4897 *       to load up translations and handle anonymous pages
4898 *   endloop
4899 *   Load up translation to any additional pages in page list not
4900 *     already handled that fit into this segment
4901 */
4902 static faultcode_t
4903 segvn_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
4904   enum fault_type type, enum seg_rw rw)
4905 {
4906     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4907     page_t *plp, *ppp, *pp;
4908     u_offset_t off;
4909     caddr_t a;
4910     struct vpage *vpage;
4911     uint_t vpprot, prot;
4912     int err;
4913     page_t *pl[FVN_GETPAGE_NUM + 1];
4914     size_t plsz, pl_alloc_sz;
4915     size_t page;
4916     ulong_t anon_index;
4917     struct anon_map *amp;
4918     int dogetpage = 0;
4919     caddr_t lpgaddr, lpgeaddr;
4920     size_t pgsz;
4921     anon_sync_obj_t cookie;
4922     int brkcow = BREAK_COW_SHARE(rw, type, svd->type);
4923
4924     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
4925     ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);
4926
4927     /*
4928     * First handle the easy stuff
4929     */
4930     if (type == F_SOFTUNLOCK) {
4931         if (rw == S_READ_NOCOW) {
4932             rw = S_READ;
4933             ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
4934         }
4935         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4936         pgsz = (seg->s_szc == 0) ? PAGESIZE :
4937             page_get_pagesize(seg->s_szc);
4938         VM_STAT_COND_ADD(pgsz > PAGESIZE, segvnmstats.fltanpages[16]);
4939         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
4940         segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
4941         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4942         return (0);
4943     }

```

```

4945 }
4946 ASSERT(svd->tr_state == SEGVN_TR_OFF ||
4947   !HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
4948 if (brkcow == 0) {
4949     if (svd->tr_state == SEGVN_TR_INIT) {
4950         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4951         if (svd->tr_state == SEGVN_TR_INIT) {
4952             ASSERT(svd->vp != NULL && svd->amp == NULL);
4953             ASSERT(svd->flags & MAP_TEXT);
4954             ASSERT(svd->type == MAP_PRIVATE);
4955             segvn_textrepl(seg);
4956             ASSERT(svd->tr_state != SEGVN_TR_INIT);
4957             ASSERT(svd->tr_state != SEGVN_TR_ON ||
4958               svd->amp != NULL);
4959         }
4960         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4961     }
4962 } else if (svd->tr_state != SEGVN_TR_OFF) {
4963     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4964
4965     if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
4966         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
4967         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4968         return (FC_PROT);
4969     }
4970
4971     if (svd->tr_state == SEGVN_TR_ON) {
4972         ASSERT(svd->vp != NULL && svd->amp != NULL);
4973         segvn_textunrepl(seg, 0);
4974         ASSERT(svd->amp == NULL &&
4975           svd->tr_state == SEGVN_TR_OFF);
4976     } else if (svd->tr_state != SEGVN_TR_OFF) {
4977         svd->tr_state = SEGVN_TR_OFF;
4978     }
4979     ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
4980     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4981 }
4982 }
4983 top:
4984 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4985
4986 /*
4987 * If we have the same protections for the entire segment,
4988 * insure that the access being attempted is legitimate.
4989 */
4990
4991 if (svd->pageprot == 0) {
4992     uint_t protchk;
4993
4994     switch (rw) {
4995     case S_READ:
4996     case S_READ_NOCOW:
4997         protchk = PROT_READ;
4998         break;
4999     case S_WRITE:
5000         protchk = PROT_WRITE;
5001         break;
5002     case S_EXEC:
5003         protchk = PROT_EXEC;
5004         break;
5005     case S_OTHER:
5006         default:
5007             protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
5008             break;
5009     }
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 }
5017
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 }
5037
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;
5067
5068     if (rw != S_READ_NOCOW) {
5069         demote = 1;
5070     }
5071
5072     if (!demote && len > PAGESIZE) {
5073         pgsz = page_get_pagesize(seg->s_szc);
5074         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
5075                         lpgeaddr);
5076         if (lpgeaddr - lpgaddr > pgsz) {
5077             demote = 1;
5078         }
5079     }
5080 }
5081
5082     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5083     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5084     if (seg->s_szc != 0) {
5085         segvn_vmpss_clrszc_cnt++;
5086         ASSERT(svd->softlockcnt == 0);
5087         err = segvn_clrszc(seg);
5088         if (err) {
5089             segvn_vmpss_clrszc_err++;
5090             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5091             return (FC_MAKE_ERR(err));
5092         }
5093     }
5094     ASSERT(seg->s_szc == 0);
5095     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5096     goto top;
5097 }
5098
5099 */
5100
5101 /* Check to see if we need to allocate an anon_map structure.
5102 */
5103 if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
5104     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5105     /*
5106      * Drop the "read" lock on the segment and acquire
5107      * the "write" version since we have to allocate the
5108      * anon_map.
5109      */
5110     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5111     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5112
5113     if (svd->amp == NULL) {
5114         svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
5115         svd->amp->a_szc = seg->s_szc;
5116     }
5117     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5118
5119     /*
5120      * Start all over again since segment protections
5121      * may have changed after we dropped the "read" lock.
5122      */
5123     goto top;
5124 }
5125
5126 */
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 }
5137
5138 amp = svd->amp;
5139
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     }
5163
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;
5168
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;
5174
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:
5201     if (svd->vpage == NULL)
5202         vpage = NULL;
5203     else
5204         vpage = &svd->vpage[page];
5205
5206     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     */
5214
5215     if ((page != 0) && filtadvice && 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;
5222
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;
5232
5233             while (pgoff > svd->offset) {
5234                 if (svd->advice != MADV_SEQUENTIAL &&
5235                     (!svd->pageadvice || (vpage &&
5236                     VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
5237                     break;
5238
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             /*
5267              * Skip pages that are free or have an
5268              * "exclusive" lock.
5269              */
5270             pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
5271             if (pp == NULL)
5272                 break;
5273
5274             /*
5275              * We don't need the page_struct_lock to test
5276              * 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)fploff, PAGESIZE,
5294             (B_DONTNEED|B_FREE|B_ASYNC),
5295             svd->cred, NULL);
5296         VN_RELEASE(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
5310     }
5311
5312     plp = pl;
5313     *plp = NULL;
5314     pl_alloc_sz = 0;
5315
5316     /*
5317     * See if we need to call VOP_GETPAGE for
5318     * *any* of the range being faulted on.
5319     * We can skip all of this work if there
5320     * was no original vnode.
5321     */
5322     if (svd->vp != NULL) {
5323         u_offset_t vp_off;
5324         size_t vp_len;
5325         struct anon *ap;
5326         vnode_t *vp;
5327
5328         vp_off = off;
5329         vp_len = len;
5330
5331         if (amp == NULL)
5332             dogetpage = 1;
5333         else {
5334             /*
5335             * Only acquire reader lock to prevent amp->ahp
5336             * from being changed. It's ok to miss pages,
5337             * hence we don't do anon_array_enter
5338             */
5339             ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5340             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 }
5349
5350 if (dogetpage) {
5351     enum seg_rw arw;
5352     struct as *as = seg->s_as;
5353
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             (bttop(len) + 1) * sizeof (page_t *);
5361         plp = kmalloc(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 == 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     }
5387
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_offset, vp_len,
5410     &vpprot, plp, plsz, seg, addr + (vp_offset - 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 */
5423 /*
5424 * N.B. at this time the plp array has all the needed non-anon
5425 * pages in addition to (possibly) having some adjacent pages.
5426 */
5427
5428 /*
5429 * Always acquire the anon_array_lock to prevent
5430 * 2 threads from allocating separate anon slots for
5431 * the same "addr".
5432 *
5433 * If this is a copy-on-write fault and we don't already
5434 * have the anon_array_lock, acquire it to prevent the
5435 * fault routine from handling multiple copy-on-write faults
5436 * on the same "addr" in the same address space.
5437 *
5438 * Only one thread should deal with the fault since after
5439 * it is handled, the other threads can acquire a translation
5440 * to the newly created private page. This prevents two or
5441 * more threads from creating different private pages for the
5442 * same fault.
5443 *
5444 * We grab "serialization" lock here if this is a MAP_PRIVATE segment
5445 * to prevent deadlock between this thread and another thread
5446 * which has soft-locked this page and wants to acquire serial_lock.
5447 * ( bug 4026339 )
5448 *
5449 * The fix for bug 4026339 becomes unnecessary when using the
5450 * locking scheme with per amp rwlock and a global set of hash
5451 * lock, anon_array_lock. If we steal a vnode page when low
5452 * on memory and upgrad the page lock through page_rename,
5453 * then the page is PAGE_HANDLED, nothing needs to be done
5454 * for this page after returning from segvn_faultpage.
5455 *
5456 * But really, the page lock should be downgraded after
5457 * the stolen page is page_rename'd.
5458 */
5459
5460 if (amp != NULL)
5461     ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5462
5463 /*
5464 * Ok, now loop over the address range and handle faults
5465 */
5466 for (a = addr; a < addr + len; a += PAGESIZE, off += PAGESIZE) {
5467     err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
5468     type, rw, brkcow);
5469     if (err) {
5470         if (amp != NULL)
5471             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 */
5490 /* Didn't get pages from the underlying fs so we're done */
5491 if (!dogetpage)
5492     goto done;
5493
5494 /*
5495 * Now handle any other pages in the list returned.
5496 * If the page can be used, load up the translations now.
5497 * Note that the for loop will only be entered if "plp"
5498 * is pointing to a non-NULL page pointer which means that
5499 * VOP_GETPAGE() was called and vpprot has been initialized.
5500 */
5501 if (svd->pageprot == 0)
5502     prot = svd->prot & vpprot;
5503
5504 /*
5505 * Large Files: diff should be unsigned value because we started
5506 * supporting > 2GB segment sizes from 2.5.1 and when a
5507 * large file of size > 2GB gets mapped to address space
5508 * the diff value can be > 2GB.
5509 */
5510
5511 for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
5512     size_t diff;
5513     struct anon *ap;
5514     int anon_index;
5515     anon_sync_obj_t cookie;
5516     int hat_flag = HAT_LOAD_ADV;
5517
5518     if (svd->flags & MAP_TEXT) {
5519         hat_flag |= HAT_LOAD_TEXT;
5520     }
5521
5522     if (pp == PAGE_HANDLED)
5523         continue;
5524
5525     if (svd->tr_state != SEGVN_TR_ON &&
5526         pp->p_offset >= svd->offset &&
5527         pp->p_offset < svd->offset + seg->s_size) {
5528
5529         diff = pp->p_offset - svd->offset;
5530
5531         /*
5532          * Large Files: Following is the assertion
5533          * validating the above cast.
5534          */
5535         ASSERT(svd->vp == pp->p_vnode);
5536
5537

```

```

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     if (amp != NULL)
5578         anon_array_exit(&cookie);
5579     page_unlock(pp);
5580 }
5581 done:
5582     if (amp != NULL)
5583         ANON_LOCK_EXIT(&amp->a_rwlock);
5584     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5585     if (pl_alloc_sz)
5586         kmem_free(plp, pl_alloc_sz);
5587     return (0);
5588 }
5589 */
5590 */
5591 /* This routine is used to start I/O on pages asynchronously. XXX it will
5592 * only create PAGESIZE pages. At fault time they will be relocated into
5593 * larger pages.
5594 */
5595 static faultcode_t
5596 segvn_faulta(struct seg *seg, caddr_t addr)
5597 {
5598     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
5599     int err;
5600     struct anon_map *amp;
5601     vnode_t *vp;
5602
5603
5604
5605
5606
5607
5608
5609
5610
5611
5612
5613
5614
5615
5616
5617
5618
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
5650
5651
5652
5653
5654
5655
5656
5657
5658
5659
5660
5661
5662
5663
5664
5665
5666
5667
5668
5669
5670
5671
5672
5673
5674
5675
5676
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
5700
5701
5702
5703
5704
5705
5706
5707
5708
5709
5710
5711
5712
5713
5714
5715
5716
5717
5718
5719
5720
5721
5722
5723
5724
5725
5726
5727
5728
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
5750
5751
5752
5753
5754
5755
5756
5757
5758
5759
5760
5761
5762
5763
5764
5765
5766
5767
5768
5769
5770
5771
5772
5773
5774
5775
5776
5777
5778
5779
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
5800
5801
5802
5803
5804
5805
5806
5807
5808
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
5850
5851
5852
5853
5854
5855
5856
5857
5858
5859
5860
5861
5862
5863
5864
5865
5866
5867
5868
5869
5870
5871
5872
5873
5874
5875
5876
5877
5878
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
5900
5901
5902
5903
5904
5905
5906
5907
5908
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
5950
5951
5952
5953
5954
5955
5956
5957
5958
5959
5960
5961
5962
5963
5964
5965
5966
5967
5968
5969
5970
5971
5972
5973
5974
5975
5976
5977
5978
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
6000
6001
6002
6003
6004
6005
6006
6007
6008
6009
6010
6011
6012
6013
6014
6015
6016
6017
6018
6019
6020
6021
6022
6023
6024
6025
6026
6027
6028
6029
6030
6031
6032
6033
6034
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
6050
6051
6052
6053
6054
6055
6056
6057
6058
6059
6060
6061
6062
6063
6064
6065
6066
6067
6068
6069
6070
6071
6072
6073
6074
6075
6076
6077
6078
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
60100
60101
60102
60103
60104
60105
60106
60107
60108
60109
60110
60111
60112
60113
60114
60115
60116
60117
60118
60119
60120
60121
60122
60123
60124
60125
60126
60127
60128
60129
60130
60131
60132
60133
60134
60135
60136
60137
60138
60139
60140
60141
60142
60143
60144
60145
60146
60147
60148
60149
60150
60151
60152
60153
60154
60155
60156
60157
60158
60159
60160
60161
60162
60163
60164
60165
60166
60167
60168
60169
60170
60171
60172
60173
60174
60175
60176
60177
60178
60179
60180
60181
60182
60183
60184
60185
60186
60187
60188
60189
60190
60191
60192
60193
60194
60195
60196
60197
60198
60199
60200
60201
60202
60203
60204
60205
60206
60207
60208
60209
60210
60211
60212
60213
60214
60215
60216
60217
60218
60219
60220
60221
60222
60223
60224
60225
60226
60227
60228
60229
60230
60231
60232
60233
60234
60235
60236
60237
60238
60239
60240
60241
60242
60243
60244
60245
60246
60247
60248
60249
60250
60251
60252
60253
60254
60255
60256
60257
60258
60259
60260
60261
60262
60263
60264
60265
60266
60267
60268
60269
60270
60271
60272
60273
60274
60275
60276
60277
60278
60279
60280
60281
60282
60283
60284
60285
60286
60287
60288
60289
60290
60291
60292
60293
60294
60295
60296
60297
60298
60299
60300
60301
60302
60303
60304
60305
60306
60307
60308
60309
60310
60311
60312
60313
60314
60315
60316
60317
60318
60319
60320
60321
60322
60323
60324
60325
60326
60327
60328
60329
60330
60331
60332
60333
60334
60335
60336
60337
60338
60339
60340
60341
60342
60343
60344
60345
60346
60347
60348
60349
60350
60351
60352
60353
60354
60355
60356
60357
60358
60359
60360
60361
60362
60363
60364
60365
60366
60367
60368
60369
60370
60371
60372
60373
60374
60375
60376
60377
60378
60379
60380
60381
60382
60383
60384
60385
60386
60387
60388
60389
60390
60391
60392
60393
60394
60395
60396
60397
60398
60399
60400
60401
60402
60403
60404
60405
60406
60407
60408
60409
60410
60411
60412
60413
60414
60415
60416
60417
60418
60419
60420
60421
60422
60423
60424
60425
60426
60427
60428
60429
60430
60431
60432
60433
60434
60435
60436
60437
60438
60439
60440
60441
60442
60443
60444
60445
60446
60447
60448
60449
60450
60451
60452
60453
60454
60455
60456
60457
60458
60459
60460
60461
60462
60463
60464
60465
60466
60467
60468
60469
60470
60471
60472
60473
60474
60475
60476
60477
60478
60479
60480
60481
60482
60483
60484
60485
60486
60487
60488
60489
60490
60491
60492
60493
60494
60495
60496
60497
60498
60499
60500
60501
60502
60503
60504
60505
60506
60507
60508
60509
60510
60511
60512
60513
60514
60515
60516
60517
60518
60519
60520
60521
60522
60523
60524
60525
60526
60527
60528
60529
60530
60531
60532
60533
60534
60535
60536
60537
60538
60539
60540
60541
60542
60543
60544
60545
60546
60547
60548
60549
60550
60551
60552
60553
60554
60555
60556
60557
60558
60559
60560
60561
60562
60563
60564
60565
60566
60567
60568
60569
60570
60571
60572
60573
60574
60575
60576
60577
60578
60579
60580
60581
60582
60583
60584
60585
60586
60587
60588
60589
60590
60591
60592
60593
60594
60595
60596
60597
60598
60599
60600
60601
60602
60603
60604
60605
60606
60607
60608
60609
60610
60611
60612
60613
60614
60615
60616
60617
60618
60619
60620
60621
60622
60623
60624
60625
60626
60627
60628
60629
60630
60631
60632
60633
60634
60635
60636
60637
60638
60639
60640
60641
60642
60643
60644
60645
60646
60647
60648
60649
60650
60651
60652
60653
60654
60655
60656
60657
60658
60659
60660
60661
60662
60663
60664
60665
60666
60667
60668
60669
60670
60671
60672
60673
60674
60675
60676
60677
60678
60679
60680
60681
60682
60683
60684
60685
60686
60687
60688
60689
60690
60691
60692
60693
60694
60695
60696
60697
60698
60699
60700
60701
60702
60703
60704
60705
60706
60707
60708
60709
60710
60711
60712
60713
60714
60715
60716
60717
60718
60719
60720
60721
60722
60723
60724
60725
60726
60727
60728
60729
60730
60731
60732
60733
60734
60735
60736
60737
60738
60739
60740
60741
60742
60743
60744
60745
60746
60747
60748
60749
60750
60751
60752
60753
60754
60755
60756
60757
60758
60759
60760
60761
60762
60763
60764
60765
60766
60767
60768
60769
60770
60771
60772
60773
60774
60775
60776
60777
60778
60779
60780
60781
60782
60783
60784
60785
60786
60787
60788
60789
60790
60791
60792
60793
60794
60795
60796
60797
60798
60799
60800
60801
60802
60803
60804
60805
60806
60807
60808
60809
60810
60811
60812
60813
60814
60815
60816
60817
60818
60819
60820
60821
60822
60823
60824
60825
60826
60827
60828
60829
60830
60831
60832
60833
60834
60835
60836
60837
60838
60839
60840
60841
60842
60843
60844
60845
60846
60847
60848
60849
60850
60851
60852
60853
60854
60855
60856
60857
60858
60859
60860
60861
60862
60863
60864
60865
60866
60867
60868
60869
60870
60871
60872
60873
60874
60875
60876
60877
60878
60879
60880
60881
60882
60883
60884
60885
60886
60887
60888
60889
60890
60891
60892
60893
60894
60895
60896
60897
60898
60899
60900
60901
60902
60903
60904
60905
60906
60907
60908
60909
60910
60911
60912
60913
60914
60915
60916
60917
60918
60919
60920
60921
60922
60923
60924
60925
60926
60927
60928
60929
60930
60931
60932
60933
60934
60935
60936
60937
60938
60939
60940
60941
60942
60943
60944
60945
60946
60947
60948
60949
60950
60951
60952
60953
60954
60955
60956
60957
60958
60959
60960
60961
60962
60963
60964
60965
60966
60967
60968
60969
60970
60971
60972
60973
60974
60975
60976
60977
60978
60979
60980
60981
60982
60983
60984
60985
60986
60987
60988
60989
60990
60991
60992
60993
60994
60995
60996
60997
60998
60999
609999
610000
610001
610002
610003
610004
610005
610006
610007
610008
610009
6100010
6100011
6100012
6100013
6100014
6100015
6100016
6100017
6100018
6100019
6100020
6100021
6100022
6100023
6100024
6100025
6100026
6100027
6100028
6100029
6100030
6100031
6100032
6100033
6100034
6100035
6100036
6100037
6100038
6100039
6100040
6100041
6100042
6100043
6100044
6100045
6100046
6100047
6100048
6100049
6100050
6100051
6100052
6100053
6100054
6100055
6100056
6100057
6100058
6100059
6100060
6100061
6100062
6100063
6100064
6100065
6100066
6100067
6100068
6100069
6100070
6100071
6100072
6100073
6100074
6100075
6100076
6100077
6100078
6100079
6100080
6100081
6100082
6100083
6100084
6100085
6100086
6100087
6100088
6100089
6100090
6100091
6100092
6100093
6100094
6100095
6100096
6100097
6100098
6100099
61000100
61000101
61000102
61000103
61000104
61000105
61000106
61000107
61000108
61000109
61000110
61000111
61000112
61000113
61000114
61000115
61000116
61000117
61000118
61000119
61000120
61000121
61000122
61000123
61000124
61000125
61000126
61000127
61000128
61000129
61000130
61000131
61000132
61000133
61000134
61000135
61000136
61000137
61000138
61000139
61000140
61000141
61000142
61000143
61000144
61000145
61000146
61000147
61000148
61000149
61000150
61000151
61000152
61000153
61000154
61000155
61000156
61000157
61000158
61000159
61000160
61000161
61000162
61000163
61000164
61000165
61000166
61000167
61000168
61000169
61000170
61000171
61000172
61000173
61000174
61000175
61000176
61000177
61000178
61000179
61000180
61000181
61000182
61000183
61000184
61000185
61000186
61000187
61000188
61000189
61000190
61000191
61000192
61000193
61000194
61000195
61000196
61000197
61000198
61000199
61000200
61000201
61000202
61000203
61000204
61000205
61000206
61000207
61000208
61000209
61000210
61000211
61000212
61000213
61000214
61000215
61000216
61000217
61000218
61000219
61000220
61000221
61000222
61000223
61000224
61000225
61000226
61000227
61000228
61000229
61000230
61000231
61000232
61000233
61000234
61000235
61000236
61000237
61000238
61000239
61000240
61000241
61000242
61000243
61000244
61000245
61000246
61000247
61000248
61000249
61000250
61000251
61000252
61000253
61000254
61000255
61000256
61000257
61000258
61000259
61000260
61000261
61000262
61000263
61000264
61000265
61000266
61000267
61000268
61000269
61000270
61000271
61000272
61000273
61000274
61000275
61000276
61000277
61000278
61000279
61000280
61000281
61000282
61000283
61000284
61000285
61000286
61000287
61000288
61000289
61000290
61000291
61000292
61000293
61000294
61000295
61000296
61000297
61000298
61000299
61000300
6
```

```

5671         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5672         return (0);
5673     }
5674
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);
5682
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         }
5691
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     }
5704
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_textunrep(seg, 0);
5717         ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
5718         unload_done = 1;
5719     }
5720
5721     if ((prot & PROT_WRITE) && svd->type == MAP_SHARED &&
5722         svd->vp != NULL && (svd->vp->v_flag & VVMEEXEC)) {
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(segvnmstats.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 }
5759
5760 /*
5761  * If it's a private mapping and we're making it writable then we
5762  * may have to reserve the additional swap space now. If we are
5763  * making writable only a part of the segment then we use its vpage
5764  * array to keep a record of the pages for which we have reserved
5765  * swap. In this case we set the pageswap field in the segment's
5766  * segvn structure to record this.
5767 */
5768
5769 /*
5770  * If it's a private mapping to a file (i.e., vp != NULL) and we're
5771  * removing write permission on the entire segment and we haven't
5772  * modified any pages, we can release the swap space.
5773 */
5774 if (svd->type == MAP_PRIVATE) {
5775     if (prot & PROT_WRITE) {
5776         if (!(svd->flags & MAP_NORESERVE) &&
5777             !(svd->swresv && svd->pageswap == 0)) {
5778             size_t sz = 0;
5779
5780             /*
5781              * Start by determining how much swap
5782              * space is required.
5783             */
5784             if (addr == seg->s_base &&
5785                 len == seg->s_size &&
5786                 svd->pageswap == 0) {
5787                 /* The whole segment */
5788                 sz = seg->s_size;
5789             } else {
5790                 /*
5791                  * Make sure that the vpage array
5792                  * exists, and make a note of the
5793                  * range of elements corresponding
5794                  * to len.
5795                 */
5796                 segvn_vpage(seg);
5797                 svp = &svd->vpage[seg_page(seg, addr)];
5798                 evp = &svd->vpage[seg_page(seg,
5799                               addr + len)];
5800
5801                 if (svd->pageswap == 0) {
5802                     /*
5803                      * 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
5822     /* Try to reserve the necessary swap. */
5823     if (anon_resv_zone(sz,
5824         seg->s_as->a_proc->p_zone) == 0) {
5825         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5826         return (IE_NOMEM);
5827     }
5828
5829     /*
5830     * Make a note of how much swap space
5831     * we've reserved.
5832     */
5833     if (svd->pageswap == 0 && sz == seg->s_size) {
5834         svd->swresv = sz;
5835     } else {
5836         ASSERT(svd->vpage != NULL);
5837         svd->swresv += sz;
5838         svd->pageswap = 1;
5839         for (cvp = svp; cvp < evp; cvp++) {
5840             if (!VPP_ISSWAPRES(cvp))
5841                 VPP_SETSWAPRES(cvp);
5842         }
5843     }
5844
5845 } else {
5846     /*
5847     * Swap space is released only if this segment
5848     * does not map anonymous memory, since read faults
5849     * on such segments still need an anon slot to read
5850     * in the data.
5851     */
5852     if (svd->swresv != 0 && svd->vp != NULL &&
5853         svd->amp == NULL && addr == seg->s_base &&
5854         len == seg->s_size && svd->pageprot == 0) {
5855         ASSERT(svd->pageswap == 0);
5856         anon_unresv_zone(svd->swresv,
5857             seg->s_as->a_proc->p_zone);
5858         svd->swresv = 0;
5859         TRACE_3(TR_FAC_VM, TR_ANON_PROC,
5860             "anon proc:%p %lu %u", seg, 0, 0);
5861     }
5862 }
5863
5864 if (addr == seg->s_base && len == seg->s_size && svd->vpage == NULL) {
5865     if (svd->prot == prot) {
5866         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5867     }
5868 }
```

```

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;
5878
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     }
5896
5897     offset = svd->offset + (uintptr_t)(addr - seg->s_base);
5898     evp = &svd->vpage[seg_page(seg, addr + len)];
5899
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++) {
5905         if (seg->s_szc != 0) {
5906             if (amp != NULL) {
5907                 anon_array_enter(amp, anon_idx,
5908                     &cookie);
5909             }
5910             if (IS_P2ALIGNED(anon_idx, pgcnt) &&
5911                 !segvn_claim_pages(seg, svp, offset,
5912                     anon_idx, prot)) {
5913                 if (amp != NULL) {
5914                     anon_array_exit(&cookie);
5915                 }
5916                 break;
5917             }
5918             if (amp != NULL) {
5919                 anon_array_exit(&cookie);
5920             }
5921             anon_idx++;
5922         } else {
5923             if (amp != NULL) {
5924                 anon_array_enter(amp, anon_idx,
5925                     &cookie);
5926                 ap = anon_get_ptr(amp->ahp, anon_idx++);
5927             }
5928
5929             if (VPP_ISPPLOCK(svp) &&
5930                 VPP_PROT(svp) != prot) {
5931                 if (amp == NULL || ap == NULL) {
5932                     if (vp == svd->vp)
```

```

5935                     off = offset;
5936             } else
5937                 swap_xlate(ap, &vp, &off);
5938             if (amp != NULL)
5939                 anon_array_exit(&cookie);
5940
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                 }
5956             }
5957             if (!page_subclaim(
5958                 pp)) {
5959                 page_unlock(pp);
5960                 break;
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);
5972
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 }
5999
6000 if (unload_done) {

```

```

6001             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6002             return (0);
6003         }
6004
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         }
6029     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6030
6031     return (0);
6032 }
6033
6034 /*
6035  * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
6036  * to determine if the seg is capable of mapping the requested szc.
6037 */
6038 static int
6039 segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
6040 {
6041     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6042     struct segvn_data *nsvd;
6043     struct anon_map *amp = svd->amp;
6044     struct seg *nseg;
6045     caddr_t eaddr = addr + len, a;
6046     size_t pgsz = page_get_pagesize(szc);
6047     pgcnt_t pgcnt = page_get_pagecnt(szc);
6048     int err;
6049     u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);
6050
6051     ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6052     ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);
6053
6054     if (seg->s_szc == szc || segvn_lpg_disable != 0) {
6055         return (0);
6056     }
6057
6058     /*
6059     * addr should always be pgsz aligned but eaddr may be misaligned if
6060     * it's at the end of the segment.
6061     *
6062     * XXX we should assert this condition since as_setpagesize() logic
6063     * guarantees it.
6064     */
6065     if (!IS_P2ALIGNED(addr, pgsz) ||

```

new/usr/src/uts/common/vm/seg\_vn.c

93

```

6067      (!IS_P2ALIGNED(eaddr, pgsz) &&
6068      eaddr != seg->s_base + seg->s_size)) {
6069          segvn_setpgsz_align_err++;
6070          return (EINVAL);
6071      }
6072  }
6073
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)) {
6077          segvn_setpgsz_anon_align_err++;
6078          return (EINVAL);
6079      }
6080  }
6081
6082  if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
6083      szc > segvn_maxpgszc) {
6084      return (EINVAL);
6085  }
6086
6087  /* paranoid check */
6088  if (svd->vp != NULL &&
6089      (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
6090      return (EINVAL);
6091  }
6092
6093  if (seg->s_szc == 0 && svd->vp != NULL &&
6094      map_addr_vacalign_check(addr, off)) {
6095      return (EINVAL);
6096  }
6097
6098  /*
6099   * Check that protections are the same within new page
6100   * size boundaries.
6101   */
6102  if (svd->pageprot) {
6103      for (a = addr; a < eaddr; a += pgsz) {
6104          if ((a + pgsz) > eaddr) {
6105              if (!sameprot(seg, a, eaddr - a)) {
6106                  return (EINVAL);
6107              }
6108          } else {
6109              if (!sameprot(seg, a, pgsz)) {
6110                  return (EINVAL);
6111              }
6112          }
6113      }
6114  }
6115
6116  /*
6117   * Since we are changing page size we first have to flush
6118   * the cache. This makes sure all the pagelock calls have
6119   * to recheck protections.
6120   */
6121  if (svd->softlockcnt > 0) {
6122      ASSERT(svd->tr_state == SEGVN_TR_OFF);
6123
6124      /*
6125       * If this is shared segment non 0 softlockcnt
6126       * means locked pages are still in use.
6127       */
6128      if (svd->type == MAP_SHARED) {
6129          return (EAGAIN);
6130      }
6131  }

```

[new/usr/src/uts/common/vm/seg\\_vn.](#)

```

6133
6134         */
6135         * Since we do have the segvn writers lock nobody can fill
6136         * the cache with entries belonging to this seg during
6137         * the purge. The flush either succeeds or we still have
6138         * pending I/Os.
6139         */
6140         segvn_purge(seg);
6141         if (svd->softlockcnt > 0) {
6142             return (EAGAIN);
6143         }
6145
6146         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6147             ASSERT(svd->amp == NULL);
6148             ASSERT(svd->tr_state == SEGVN_TR_OFF);
6149             hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6150                             HAT_REGION_TEXT);
6151             svd->rcookie = HAT_INVALID_REGION_COOKIE;
6152         } else if (svd->tr_state == SEGVN_TR_INIT) {
6153             svd->tr_state = SEGVN_TR_OFF;
6154         } else if (svd->tr_state == SEGVN_TR_ON) {
6155             ASSERT(svd->amp != NULL);
6156             segvn_textunrepl(seg, 1);
6157             ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6158             amp = NULL;
6159         }
6160
6161         /*
6162         * Operation for sub range of existing segment.
6163         */
6164         if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6165             if (szc < seg->s_szc) {
6166                 VM_STAT_ADD(segvnvmstats.demoterange[2]);
6167                 err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6168                 if (err == 0) {
6169                     return (IE_RETRY);
6170                 }
6171                 if (err == ENOMEM) {
6172                     return (IE_NOMEM);
6173                 }
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             }
6182             return (IE_RETRY);
6183         }
6184         if (eaddr != (seg->s_base + seg->s_size)) {
6185             /* eaddr is szc aligned */
6186             (void) segvn_split_seg(seg, eaddr);
6187         }
6188         return (IE_RETRY);
6189     }
6190
6191     /*
6192     * Break any low level sharing and reset seg->s_szc to 0.
6193     */
6194     if ((err = segvn_clrszc(seg)) != 0) {
6195         if (err == ENOMEM) {
6196             err = IE_NOMEM;
6197         }
6198     }

```

```

6199     ASSERT(seg->s_szc == 0);

6200     /*
6201      * If the end of the current segment is not pgsz aligned
6202      * then attempt to concatenate with the next segment.
6203      */
6204     if (!IS_P2ALIGNED(eaddr, pgsz)) {
6205         nseg = AS_SEGNEXT(seg->s_as, seg);
6206         if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6207             return (ENOMEM);
6208         }
6209         if (nseg->s_ops != &segvn_ops) {
6210             return (EINVAL);
6211         }
6212         nsvd = (struct segvn_data *)nseg->s_data;
6213         if (nsvd->softlockcnt > 0) {
6214             /*
6215              * If this is shared segment non 0 softlockcnt
6216              * means locked pages are still in use.
6217              */
6218             if (nsvd->type == MAP_SHARED) {
6219                 return (EAGAIN);
6220             }
6221             segvn_purge(nseg);
6222             if (nsvd->softlockcnt > 0) {
6223                 return (EAGAIN);
6224             }
6225         }
6226         err = segvn_clrszc(nseg);
6227         if (err == ENOMEM) {
6228             err = IE_NOMEM;
6229         }
6230         if (err != 0) {
6231             return (err);
6232         }
6233         ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6234         err = segvn_concat(seg, nseg, 1);
6235         if (err == -1) {
6236             return (EINVAL);
6237         }
6238         if (err == -2) {
6239             return (IE_NOMEM);
6240         }
6241         return (IE_RETRY);
6242     }

6243     /*
6244      * May need to re-align anon array to
6245      * new szc.
6246      */
6247     if (amp != NULL) {
6248         if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6249             struct anon_hdr *nahp;

6250             ASSERT(svd->type == MAP_PRIVATE);

6251             ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6252             ASSERT(amp->refcnt == 1);
6253             nahp = anon_create(bttop(amp->size), ANON_NOSLEEP);
6254             if (nahp == NULL) {
6255                 ANON_LOCK_EXIT(&amp->a_rwlock);
6256                 return (IE_NOMEM);
6257             }
6258             if (anon_copy_ptr(amp->ahp, svd->anon_index,
6259                             nahp, 0, bttop(seg->s_size), ANON_NOSLEEP)) {
6260                 anon_release(nahp, bttop(amp->size));
6261             }
6262         }
6263     }

```

```

6265     ANON_LOCK_EXIT(&amp->a_rwlock);
6266     return (IE_NOMEM);
6267 }
6268 anon_release(amp->ahp, bttop(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_set vnode_mpss(svd->vp);
6305 }

6306 if (amp != NULL) {
6307     ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6308     if (svd->type == MAP_PRIVATE) {
6309         amp->a_szc = szc;
6310     } else if (sdc > amp->a_szc) {
6311         amp->a_szc = szc;
6312     }
6313     ANON_LOCK_EXIT(&amp->a_rwlock);
6314 }
6315 }

6316 seg->s_szc = szc;

6317 return (0);
6318 }

6319 static int
6320 segvn_clrszc(struct seg *seg)
6321 {
6322     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6323     struct anon_map *amp = svd->amp;
6324     size_t pgsz;
6325     pgcnt_t pages;
6326     int err = 0;
6327     caddr_t a = seg->s_base;
6328 
```

```

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     }

6349     if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6350         ASSERT(svd->amp == NULL);
6351         ASSERT(svd->tr_state == SEGVN_TR_OFF);
6352         hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6353             HAT_REGION_TEXT);
6354         svd->rcookie = HAT_INVALID_REGION_COOKIE;
6355     } else if (svd->tr_state == SEGVN_TR_ON) {
6356         ASSERT(svd->amp != NULL);
6357         segvn_textunrep(seg, 1);
6358         ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6359         amp = NULL;
6360     } else {
6361         if (svd->tr_state != SEGVN_TR_OFF) {
6362             ASSERT(svd->tr_state == SEGVN_TR_INIT);
6363             svd->tr_state = SEGVN_TR_OFF;
6364         }
6365     }

6366     /*
6367      * do HAT_UNLOAD_UNMAP since we are changing the pagesize.
6368      * unload argument is 0 when we are freeing the segment
6369      * and unload was already done.
6370      */
6371     hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
6372                 HAT_UNLOAD_UNMAP);
6373
6374     if (amp == NULL || svd->type == MAP_SHARED) {
6375         seg->s_szc = 0;
6376         return (0);
6377     }

6378     pgsz = page_get_pagesize(seg->s_szc);
6379     pages = bttop(pgsz);

6380     /*
6381      * XXX anon rwlock is not really needed because this is a
6382      * private segment and we are writers.
6383      */
6384     ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);

6385     for (; a < ea; a += pgsz, an_idx += pages) {
6386         if ((oldap = anon_get_ptr(amp->ahp, an_idx)) != NULL) {
6387             ASSERT(vpage != NULL || svd->pageprot == 0);
6388             if (vpage != NULL) {
6389                 ASSERT(sameprot(seg, a, pgsz));
6390                 prot = VPP_PROT(vpage);
6391                 pageflag = VPP_ISPLOCK(vpage) ? LOCK_PAGE : 0;
6392             }
6393         }
6394     }
6395 }
6396

```

```

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)) != 0) {
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     vpage = (vpage == NULL) ? NULL : vpage + pages;
6425
6426     }
6427     amp->a_szc = 0;
6428     seg->s_szc = 0;
6429 out:
6430     ANON_LOCK_EXIT(&amp->a_rwlock);
6431     return (err);
6432 }
6433 }

6434 static int
6435 segvn_claim_pages(
6436     struct seg *seg,
6437     struct vpage *svp,
6438     u_offset_t off,
6439     ulong_t anon_idx,
6440     uint_t prot)
6441 {
6442     pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
6443     size_t ppasize = (pgcnt + 1) * sizeof (page_t *);
6444     page_t **ppa;
6445     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6446     struct anon_map *amp = svd->amp;
6447     struct vpage *evp = svp + pgcnt;
6448     caddr_t addr = ((uintptr_t)(svp - svd->vpage) << PAGESHIFT)
6449             + seg->s_base;
6450     struct anon *ap;
6451     struct vnode *vp = svd->vp;
6452     page_t *pp;
6453     pgcnt_t pg_idx, i;
6454     int err = 0;
6455     anoff_t aoff;
6456     int anon = (amp != NULL) ? 1 : 0;
6457
6458     ASSERT(svd->type == MAP_PRIVATE);
6459     ASSERT(svd->vpage != NULL);
6460     ASSERT(seg->s_szc != 0);
6461     ASSERT(IS_P2ALIGNED(pgcnt, pgcnt));
6462

```

```

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_ISPLOCK(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     /* Find each large page within ppa, and adjust its claim */
6510
6511     /* Does ppa cover a single large page? */
6512     if (ppa[0]->p_szc == seg->s_szc) {
6513         if (prot & PROT_WRITE)
6514             err = page_addclaim_pages(ppa);
6515         else
6516             err = page_subclaim_pages(ppa);
6517     } else {
6518         for (i = 0; ppa[i]; i += pgcnt) {
6519             ASSERT(IS_P2ALIGNED(page_pptonum(ppa[i]), pgcnt));
6520             if (prot & PROT_WRITE)
6521                 err = page_addclaim_pages(&ppa[i]);
6522             else
6523                 err = page_subclaim_pages(&ppa[i]);
6524             if (err == 0)
6525                 break;
6526         }
6527     }
6528 }
```

```

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     }
6537 }
6538 */
6539 /* Returns right (upper address) segment if split occurred.
6540 * If the address is equal to the beginning or end of its segment it returns
6541 * the current segment.
6542 */
6543 static struct seg *
6544 segvn_split_seg(struct seg *seg, caddr_t addr)
6545 {
6546     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6547     struct seg *nseg;
6548     size_t nsize;
6549     struct segvn_data *nsvd;
6550
6551     ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6552     ASSERT(svd->tr_state == SEGVN_TR_OFF);
6553
6554     ASSERT(addr >= seg->s_base);
6555     ASSERT(addr <= seg->s_base + seg->s_size);
6556     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
6557
6558     if (addr == seg->s_base || addr == seg->s_base + seg->s_size)
6559         return (seg);
6560
6561     nsize = seg->s_base + seg->s_size - addr;
6562     seg->s_size = addr - seg->s_base;
6563     nseg = seg_alloc(seg->s_as, addr, nsize);
6564     ASSERT(nseg != NULL);
6565     nseg->s_ops = seg->s_ops;
6566     nsvd = kmem_cache_alloc(segvn_cache, KM_SLEEP);
6567     nseg->s_data = (void *)nsvd;
6568     nseg->s_szc = seg->s_szc;
6569     *nsvd = *svd;
6570     ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6571     nsvd->seg = nseg;
6572     rw_init(&nsvd->lock, NULL, RW_DEFAULT, NULL);
6573
6574     if (nsvd->vp != NULL) {
6575         VN_HOLD(nsvd->vp);
6576         nsvd->offset = svd->offset +
6577             (uintptr_t)(nseg->s_base - seg->s_base);
6578         if (nsvd->type == MAP_SHARED)
6579             lgrp_shm_policy_init(NULL, nsvd->vp);
6580     } else {
6581         /*
6582          * The offset for an anonymous segment has no significance in
6583          * terms of an offset into a file. If we were to use the above
6584          * calculation instead, the structures read out of
6585          * /proc/<pid>/xmap would be more difficult to decipher since
6586          * it would be unclear whether two seemingly contiguous
6587          * prxmap_t structures represented different segments or a
6588          * single segment that had been split up into multiple prxmap_t
6589          * structures (e.g. if some part of the segment had not yet
6590          * been faulted in).
6591          */
6592         nsvd->offset = 0;
6593     }
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), nsyd->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         nsyd->amp = namp;
6632         nsyd->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 == nsyd->amp);
6637         ASSERT(seg->s_szc <= svd->amp->a_szc);
6638         nsyd->anon_index = svd->anon_index + seg_pages(seg);
6639         ASSERT(IS_P2ALIGNED(nsyd->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;

6656         ASSERT(svd->amp);
6657         oswresv = svd->swresv;
6658         svd->swresv = ptob(anon_pages(svd->amp->ahp,
6659             svd->anon_index, btop(seg->s_size)));
6660     }
6661     }
6662     ASSERT(svd->softlockcnt == 0);
6663     ASSERT(svd->softlockcnt_sbase == 0);
6664     ASSERT(svd->softlockcnt_send == 0);
6665     crhold(svd->cred);
6666     if (svd->vpage != NULL) {
6667         size_t bytes = vpgtob(seg_pages(seg));
6668         size_t nbytes = vpgtob(seg_pages(nseg));
6669         struct vpage *ovpage = svd->vpage;
6670         svd->vpage = kmem_alloc(bytes, KM_SLEEP);
6671         bcopy(ovpage, svd->vpage, bytes);
6672         nsyd->vpage = kmem_alloc(nbytes, KM_SLEEP);
6673         bcopy(ovpage + seg_pages(seg), nsyd->vpage, nbytes);
6674         kmem_free(ovpage, bytes + nbytes);
6675     }
6676     if (svd->amp != NULL && svd->type == MAP_PRIVATE) {
6677         struct anon_map *oamp = svd->amp, *namp;
6678         struct anon_hdr *nahp;
6679         ANON_LOCK_ENTER(&oamp->a_rwlock, RW_WRITER);
6680         ASSERT(oamp->refcnt == 1);
6681         nahp = anon_create(btop(seg->s_size), ANON_SLEEP);
6682         (void) anon_copy_ptr(oamp->ahp, svd->anon_index,
6683             nahp, 0, btop(seg->s_size), ANON_SLEEP);
6684         anon_release(oamp->ahp, btop(oamp->size));
6685         oamp->ahp = nahp;
6686         oamp->size = seg->s_size;
6687         svd->anon_index = 0;
6688         nsyd->amp = namp;
6689         nsyd->anon_index = 0;
6690         ANON_LOCK_EXIT(&oamp->a_rwlock);
6691     } else if (svd->amp != NULL) {
6692         pgcnt_t pgcnt = page_get_pagecnt(seg->s_szc);
6693         ASSERT(svd->amp == nsyd->amp);
6694         ASSERT(seg->s_szc <= svd->amp->a_szc);
6695         nsyd->anon_index = svd->anon_index + seg_pages(seg);
6696         ASSERT(IS_P2ALIGNED(nsyd->anon_index, pgcnt));
6697         ANON_LOCK_ENTER(&svd->amp->a_rwlock, RW_WRITER);
6698         svd->amp->refcnt++;
6699         ANON_LOCK_EXIT(&svd->amp->a_rwlock);
6700     }
6701     if (svd->swresv) {
6702         size_t pgsz;
6703         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6704         int err;
6705         uint_t szc = seg->s_szc;
6706         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));
6716         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
6717         ASSERT(flag == SDR_RANGE || eaddr < lpgeaddr || addr > lpgaddr);
6718         if (flag == SDR_RANGE) {
6719             /* demote entire range */
6720             badseg1 = nseg = segvn_split_seg(seg, lpgaddr);
6721             (void) segvn_split_seg(nseg, lpgeaddr);
6722             ASSERT(badseg1->s_base == lpgaddr);
6723             ASSERT(badseg1->s_size == lpgeaddr - lpgaddr);
6724         } else if (addr != lpgaddr) {
6725             ASSERT(flag == SDR_END);
6726         }
6727     }
6728     if (svd->softlockcnt == 0) {
6729         crhold(svd->cred);
6730     }
6731     if (svd->softlockcnt_sbase == 0) {
6732         crhold(svd->cred);
6733     }
6734     if (svd->softlockcnt_send == 0) {
6735         crhold(svd->cred);
6736     }
6737     if (svd->vpage != NULL) {
6738         if (svd->vpage->vrefcnt == 0) {
6739             vpgtob(seg_pages(seg));
6740             vpgtob(seg_pages(nseg));
6741             kmem_free(svd->vpage, bytes + nbytes);
6742         }
6743     }
6744     if (svd->amp != NULL && svd->type == MAP_PRIVATE) {
6745         if (svd->amp->refcnt == 0) {
6746             oamp->refcnt = 1;
6747             oamp->ahp->refcnt = 1;
6748             oamp->ahp->anon_index = 0;
6749             oamp->ahp->size = seg->s_size;
6750             oamp->ahp->softlockcnt = 0;
6751             oamp->ahp->softlockcnt_sbase = 0;
6752             oamp->ahp->softlockcnt_send = 0;
6753             oamp->ahp->vrefcnt = 0;
6754             oamp->ahp->vpage = NULL;
6755             oamp->ahp->vsize = 0;
6756             oamp->ahp->vbytes = 0;
6757             oamp->ahp->vpages = 0;
6758             oamp->ahp->vrefcnt = 0;
6759             oamp->ahp->vsize = 0;
6760             oamp->ahp->vbytes = 0;
6761             oamp->ahp->vpages = 0;
6762             oamp->ahp->vrefcnt = 0;
6763             oamp->ahp->vsize = 0;
6764             oamp->ahp->vbytes = 0;
6765             oamp->ahp->vpages = 0;
6766             oamp->ahp->vrefcnt = 0;
6767             oamp->ahp->vsize = 0;
6768             oamp->ahp->vbytes = 0;
6769             oamp->ahp->vpages = 0;
6770             oamp->ahp->vrefcnt = 0;
6771             oamp->ahp->vsize = 0;
6772             oamp->ahp->vbytes = 0;
6773             oamp->ahp->vpages = 0;
6774             oamp->ahp->vrefcnt = 0;
6775             oamp->ahp->vsize = 0;
6776             oamp->ahp->vbytes = 0;
6777             oamp->ahp->vpages = 0;
6778             oamp->ahp->vrefcnt = 0;
6779             oamp->ahp->vsize = 0;
6780             oamp->ahp->vbytes = 0;
6781             oamp->ahp->vpages = 0;
6782             oamp->ahp->vrefcnt = 0;
6783             oamp->ahp->vsize = 0;
6784             oamp->ahp->vbytes = 0;
6785             oamp->ahp->vpages = 0;
6786             oamp->ahp->vrefcnt = 0;
6787             oamp->ahp->vsize = 0;
6788             oamp->ahp->vbytes = 0;
6789             oamp->ahp->vpages = 0;
6790             oamp->ahp->vrefcnt = 0;
6791             oamp->ahp->vsize = 0;
6792             oamp->ahp->vbytes = 0;
6793             oamp->ahp->vpages = 0;
6794             oamp->ahp->vrefcnt = 0;
6795             oamp->ahp->vsize = 0;
6796             oamp->ahp->vbytes = 0;
6797             oamp->ahp->vpages = 0;
6798             oamp->ahp->vrefcnt = 0;
6799             oamp->ahp->vsize = 0;
6800             oamp->ahp->vbytes = 0;
6801             oamp->ahp->vpages = 0;
6802             oamp->ahp->vrefcnt = 0;
6803             oamp->ahp->vsize = 0;
6804             oamp->ahp->vbytes = 0;
6805             oamp->ahp->vpages = 0;
6806             oamp->ahp->vrefcnt = 0;
6807             oamp->ahp->vsize = 0;
6808             oamp->ahp->vbytes = 0;
6809             oamp->ahp->vpages = 0;
6810             oamp->ahp->vrefcnt = 0;
6811             oamp->ahp->vsize = 0;
6812             oamp->ahp->vbytes = 0;
6813             oamp->ahp->vpages = 0;
6814             oamp->ahp->vrefcnt = 0;
6815             oamp->ahp->vsize = 0;
6816             oamp->ahp->vbytes = 0;
6817             oamp->ahp->vpages = 0;
6818             oamp->ahp->vrefcnt = 0;
6819             oamp->ahp->vsize = 0;
6820             oamp->ahp->vbytes = 0;
6821             oamp->ahp->vpages = 0;
6822             oamp->ahp->vrefcnt = 0;
6823             oamp->ahp->vsize = 0;
6824             oamp->ahp->vbytes = 0;
6825             oamp->ahp->vpages = 0;
6826             oamp->ahp->vrefcnt = 0;
6827             oamp->ahp->vsize = 0;
6828             oamp->ahp->vbytes = 0;
6829             oamp->ahp->vpages = 0;
6830             oamp->ahp->vrefcnt = 0;
6831             oamp->ahp->vsize = 0;
6832             oamp->ahp->vbytes = 0;
6833             oamp->ahp->vpages = 0;
6834             oamp->ahp->vrefcnt = 0;
6835             oamp->ahp->vsize = 0;
6836             oamp->ahp->vbytes = 0;
6837             oamp->ahp->vpages = 0;
6838             oamp->ahp->vrefcnt = 0;
6839             oamp->ahp->vsize = 0;
6840             oamp->ahp->vbytes = 0;
6841             oamp->ahp->vpages = 0;
6842             oamp->ahp->vrefcnt = 0;
6843             oamp->ahp->vsize = 0;
6844             oamp->ahp->vbytes = 0;
6845             oamp->ahp->vpages = 0;
6846             oamp->ahp->vrefcnt = 0;
6847             oamp->ahp->vsize = 0;
6848             oamp->ahp->vbytes = 0;
6849             oamp->ahp->vpages = 0;
6850             oamp->ahp->vrefcnt = 0;
6851             oamp->ahp->vsize = 0;
6852             oamp->ahp->vbytes = 0;
6853             oamp->ahp->vpages = 0;
6854             oamp->ahp->vrefcnt = 0;
6855             oamp->ahp->vsize = 0;
6856             oamp->ahp->vbytes = 0;
6857             oamp->ahp->vpages = 0;
6858             oamp->ahp->vrefcnt = 0;
6859             oamp->ahp->vsize = 0;
6860             oamp->ahp->vbytes = 0;
6861             oamp->ahp->vpages = 0;
6862             oamp->ahp->vrefcnt = 0;
6863             oamp->ahp->vsize = 0;
6864             oamp->ahp->vbytes = 0;
6865             oamp->ahp->vpages = 0;
6866             oamp->ahp->vrefcnt = 0;
6867             oamp->ahp->vsize = 0;
6868             oamp->ahp->vbytes = 0;
6869             oamp->ahp->vpages = 0;
6870             oamp->ahp->vrefcnt = 0;
6871             oamp->ahp->vsize = 0;
6872             oamp->ahp->vbytes = 0;
6873             oamp->ahp->vpages = 0;
6874             oamp->ahp->vrefcnt = 0;
6875             oamp->ahp->vsize = 0;
6876             oamp->ahp->vbytes = 0;
6877             oamp->ahp->vpages = 0;
6878             oamp->ahp->vrefcnt = 0;
6879             oamp->ahp->vsize = 0;
6880             oamp->ahp->vbytes = 0;
6881             oamp->ahp->vpages = 0;
6882             oamp->ahp->vrefcnt = 0;
6883             oamp->ahp->vsize = 0;
6884             oamp->ahp->vbytes = 0;
6885             oamp->ahp->vpages = 0;
6886             oamp->ahp->vrefcnt = 0;
6887             oamp->ahp->vsize = 0;
6888             oamp->ahp->vbytes = 0;
6889             oamp->ahp->vpages = 0;
6890             oamp->ahp->vrefcnt = 0;
6891             oamp->ahp->vsize = 0;
6892             oamp->ahp->vbytes = 0;
6893             oamp->ahp->vpages = 0;
6894             oamp->ahp->vrefcnt = 0;
6895             oamp->ahp->vsize = 0;
6896             oamp->ahp->vbytes = 0;
6897             oamp->ahp->vpages = 0;
6898             oamp->ahp->vrefcnt = 0;
6899             oamp->ahp->vsize = 0;
6900             oamp->ahp->vbytes = 0;
6901             oamp->ahp->vpages = 0;
6902             oamp->ahp->vrefcnt = 0;
6903             oamp->ahp->vsize = 0;
6904             oamp->ahp->vbytes = 0;
6905             oamp->ahp->vpages = 0;
6906             oamp->ahp->vrefcnt = 0;
6907             oamp->ahp->vsize = 0;
6908             oamp->ahp->vbytes = 0;
6909             oamp->ahp->vpages = 0;
6910             oamp->ahp->vrefcnt = 0;
6911             oamp->ahp->vsize = 0;
6912             oamp->ahp->vbytes = 0;
6913             oamp->ahp->vpages = 0;
6914             oamp->ahp->vrefcnt = 0;
6915             oamp->ahp->vsize = 0;
6916             oamp->ahp->vbytes = 0;
6917             oamp->ahp->vpages = 0;
6918             oamp->ahp->vrefcnt = 0;
6919             oamp->ahp->vsize = 0;
6920             oamp->ahp->vbytes = 0;
6921             oamp->ahp->vpages = 0;
6922             oamp->ahp->vrefcnt = 0;
6923             oamp->ahp->vsize = 0;
6924             oamp->ahp->vbytes = 0;
6925             oamp->ahp->vpages = 0;
6926             oamp->ahp->vrefcnt = 0;
6927             oamp->ahp->vsize = 0;
6928             oamp->ahp->vbytes = 0;
6929             oamp->ahp->vpages = 0;
6930             oamp->ahp->vrefcnt = 0;
6931             oamp->ahp->vsize = 0;
6932             oamp->ahp->vbytes = 0;
6933             oamp->ahp->vpages = 0;
6934             oamp->ahp->vrefcnt = 0;
6935             oamp->ahp->vsize = 0;
6936             oamp->ahp->vbytes = 0;
6937             oamp->ahp->vpages = 0;
6938             oamp->ahp->vrefcnt = 0;
6939             oamp->ahp->vsize = 0;
6940             oamp->ahp->vbytes = 0;
6941             oamp->ahp->vpages = 0;
6942             oamp->ahp->vrefcnt = 0;
6943             oamp->ahp->vsize = 0;
6944             oamp->ahp->vbytes = 0;
6945             oamp->ahp->vpages = 0;
6946             oamp->ahp->vrefcnt = 0;
6947             oamp->ahp->vsize = 0;
6948             oamp->ahp->vbytes = 0;
6949             oamp->ahp->vpages = 0;
6950             oamp->ahp->vrefcnt = 0;
6951             oamp->ahp->vsize = 0;
6952             oamp->ahp->vbytes = 0;
6953             oamp->ahp->vpages = 0;
6954             oamp->ahp->vrefcnt = 0;
6955             oamp->ahp->vsize = 0;
6956             oamp->ahp->vbytes = 0;
6957             oamp->ahp->vpages = 0;
6958             oamp->ahp->vrefcnt = 0;
6959             oamp->ahp->vsize = 0;
6960             oamp->ahp->vbytes = 0;
6961             oamp->ahp->vpages = 0;
6962             oamp->ahp->vrefcnt = 0;
6963             oamp->ahp->vsize = 0;
6964             oamp->ahp->vbytes = 0;
6965             oamp->ahp->vpages = 0;
6966             oamp->ahp->vrefcnt = 0;
6967             oamp->ahp->vsize = 0;
6968             oamp->ahp->vbytes = 0;
6969             oamp->ahp->vpages = 0;
6970             oamp->ahp->vrefcnt = 0;
6971             oamp->ahp->vsize = 0;
6972             oamp->ahp->vbytes = 0;
6973             oamp->ahp->vpages = 0;
6974             oamp->ahp->vrefcnt = 0;
6975             oamp->ahp->vsize = 0;
6976             oamp->ahp->vbytes = 0;
6977             oamp->ahp->vpages = 0;
6978             oamp->ahp->vrefcnt = 0;
6979             oamp->ahp->vsize = 0;
6980             oamp->ahp->vbytes = 0;
6981             oamp->ahp->vpages = 0;
6982             oamp->ahp->vrefcnt = 0;
6983             oamp->ahp->vsize = 0;
6984             oamp->ahp->vbytes = 0;
6985             oamp->ahp->vpages = 0;
6986             oamp->ahp->vrefcnt = 0;
6987             oamp->ahp->vsize = 0;
6988             oamp->ahp->vbytes = 0;
6989             oamp->ahp->vpages = 0;
6990             oamp->ahp->vrefcnt = 0;
6991             oamp->ahp->vsize = 0;
6992             oamp->ahp->vbytes = 0;
6993             oamp->ahp->vpages = 0;
6994             oamp->ahp->vrefcnt = 0;
6995             oamp->ahp->vsize = 0;
6996             oamp->ahp->vbytes = 0;
6997             oamp->ahp->vpages = 0;
6998             oamp->ahp->vrefcnt = 0;
6999             oamp->ahp->vsize = 0;
7000             oamp->ahp->vbytes = 0;
7001             oamp->ahp->vpages = 0;
7002             oamp->ahp->vrefcnt = 0;
7003             oamp->ahp->vsize = 0;
7004             oamp->ahp->vbytes = 0;
7005             oamp->ahp->vpages = 0;
7006             oamp->ahp->vrefcnt = 0;
7007             oamp->ahp->vsize = 0;
7008             oamp->ahp->vbytes = 0;
7009             oamp->ahp->vpages = 0;
7010             oamp->ahp->vrefcnt = 0;
7011             oamp->ahp->vsize = 0;
7012             oamp->ahp->vbytes = 0;
7013             oamp->ahp->vpages = 0;
7014             oamp->ahp->vrefcnt = 0;
7015             oamp->ahp->vsize = 0;
7016             oamp->ahp->vbytes = 0;
7017             oamp->ahp->vpages = 0;
7018             oamp->ahp->vrefcnt = 0;
7019             oamp->ahp->vsize = 0;
7020             oamp->ahp->vbytes = 0;
7021             oamp->ahp->vpages = 0;
7022             oamp->ahp->vrefcnt = 0;
7023             oamp->ahp->vsize = 0;
7024             oamp->ahp->vbytes = 0;
7025             oamp->ahp->vpages = 0;
7026             oamp->ahp->vrefcnt = 0;
7027             oamp->ahp->vsize = 0;
7028             oamp->ahp->vbytes = 0;
7029             oamp->ahp->vpages = 0;
7030             oamp->ahp->vrefcnt = 0;
7031             oamp->ahp->vsize = 0;
7032             oamp->ahp->vbytes = 0;
7033             oamp->ahp->vpages = 0;
7034             oamp->ahp->vrefcnt = 0;
7035             oamp->ahp->vsize = 0;
7036             oamp->ahp->vbytes = 0;
7037             oamp->ahp->vpages = 0;
7038             oamp->ahp->vrefcnt = 0;
7039             oamp->ahp->vsize = 0;
7040             oamp->ahp->vbytes = 0;
7041             oamp->ahp->vpages = 0;
7042             oamp->ahp->vrefcnt = 0;
7043             oamp->ahp->vsize = 0;
7044             oamp->ahp->vbytes = 0;
7045             oamp->ahp->vpages = 0;
7046             oamp->ahp->vrefcnt = 0;
7047             oamp->ahp->vsize = 0;
7048             oamp->ahp->vbytes = 0;
7049             oamp->ahp->vpages = 0;
7050             oamp->ahp->vrefcnt = 0;
7051             oamp->ahp->vsize = 0;
7052             oamp->ahp->vbytes = 0;
7053             oamp->ahp->vpages = 0;
7054             oamp->ahp->vrefcnt = 0;
7055             oamp->ahp->vsize = 0;
7056             oamp->ahp->vbytes = 0;
7057             oamp->ahp->vpages = 0;
7058             oamp->ahp->vrefcnt = 0;
7059             oamp->ahp->vsize = 0;
7060             oamp->ahp->vbytes = 0;
7061             oamp->ahp->vpages = 0;
7062             oamp->ahp->vrefcnt = 0;
7063             oamp->ahp->vsize = 0;
7064             oamp->ahp->vbytes = 0;
7065             oamp->ahp->vpages = 0;
7066             oamp->ahp->vrefcnt = 0;
7067             oamp->ahp->vsize = 0;
7068             oamp->ahp->vbytes = 0;
7069             oamp->ahp->vpages = 0;
7070             oamp->ahp->vrefcnt = 0;
7071             oamp->ahp->vsize = 0;
7072             oamp->ahp->vbytes = 0;
7073             oamp->ahp->vpages = 0;
7074             oamp->ahp->vrefcnt = 0;
7075             oamp->ahp->vsize = 0;
7076             oamp->ahp->vbytes = 0;
7077             oamp->ahp->vpages = 0;
7078             oamp->ahp->vrefcnt = 0;
7079             oamp->ahp->vsize = 0;
7080             oamp->ahp->vbytes = 0;
7081             oamp->ahp->vpages = 0;
7082             oamp->ahp->vrefcnt = 0;
7083             oamp->ahp->vsize = 0;
7084             oamp->ahp->vbytes = 0;
7085             oamp->ahp->vpages = 0;
7086             oamp->ahp->vrefcnt = 0;
7087             oamp->ahp->vsize = 0;
7088             oamp->ahp->vbytes = 0;
7089             oamp->ahp->vpages = 0;
7090             oamp->ahp->vrefcnt = 0;
7091             oamp->ahp->vsize = 0;
7092             oamp->ahp->vbytes = 0;
7093             oamp->ahp->vpages = 0;
7094             oamp->ahp->vrefcnt = 0;
7095             oamp->ahp->vsize = 0;
7096             oamp->ahp->vbytes = 0;
7097             oamp->ahp->vpages = 0;
7098             oamp->ahp->vrefcnt = 0;
7099             oamp->ahp->vsize = 0;
7100             oamp->ahp->vbytes = 0;
7101             oamp->ahp->vpages = 0;
7102             oamp->ahp->vrefcnt = 0;
7103             oamp->ahp->vsize = 0;
7104             oamp->ahp->vbytes = 0;
7105             oamp->ahp->vpages = 0;
7106             oamp->ahp->vrefcnt = 0;
7107             oamp->ahp->vsize = 0;
7108             oamp->ahp->vbytes = 0;
7109             oamp->ahp->vpages = 0;
7110             oamp->ahp->vrefcnt = 0;
7111             oamp->ahp->vsize = 0;
7112             oamp->ahp->vbytes = 0;
7113             oamp->ahp->vpages = 0;
7114             oamp->ahp->vrefcnt = 0;
7115             oamp->ahp->vsize = 0;
7116             oamp->ahp->vbytes = 0;
7117             oamp->ahp->vpages = 0;
7118             oamp->ahp->vrefcnt = 0;
7119             oamp->ahp->vsize = 0;
7120             oamp->ahp->vbytes = 0;
7121             oamp->ahp->vpages = 0;
7122             oamp->ahp->vrefcnt = 0;
7123             oamp->ahp->vsize = 0;
7124             oamp->ahp->vbytes = 0;
7125             oamp->ahp->vpages = 0;
7126             oamp->ahp->vrefcnt = 0;
7127             oamp->ahp->vsize = 0;
7128             oamp->ahp->vbytes = 0;
7129             oamp->ahp->vpages = 0;
7130             oamp->ahp->vrefcnt = 0;
7131             oamp->ahp->vsize = 0;
7132             oamp->ahp->vbytes = 0;
7133             oamp->ahp->vpages = 0;
7134             oamp->ahp->vrefcnt = 0;
7135             oamp->ahp->vsize = 0;
7136             oamp->ahp->vbytes = 0;
7137             oamp->ahp->vpages = 0;
7138             oamp->ahp->vrefcnt = 0;
7139             oamp->ahp->vsize = 0;
7140             oamp->ahp->vbytes = 0;
7141             oamp->ahp->vpages = 0;
7142             oamp->ahp->vrefcnt = 0;
7143             oamp->ahp->vsize = 0;
7144             oamp->ahp->vbytes = 0;
7145             oamp->ahp->vpages = 0;
7146             oamp->ahp->vrefcnt = 0;
7147             oamp->ahp->vsize = 0;
7148             oamp->ahp->vbytes = 0;
7149             oamp->ahp->vpages = 0;
7150             oamp->ahp->vrefcnt = 0;
7151             oamp->ahp->vsize = 0;
715
```

```

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     ASSERT(flag == SDR_END);
6747     ASSERT(eaddr < lpgeaddr);
6748     badseg1 = nseg = segvn_split_seg(seg, lpgeaddr - pgsz);
6749     (void) segvn_split_seg(nseg, lpgeaddr);
6750     ASSERT(badseg1->s_base == lpgeaddr - pgsz);
6751     ASSERT(badseg1->s_size == pgsz);
6752
6753 }
6754
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);
6766
6767 if (szc > 1 && (tszvvec = P2PHASE(szvvec, 1 << szc)) > 1) {
6768     uint_t tszc = highbit(tszvvec) - 1;
6769     caddr_t ta = MAX(addr, badseg1->s_base);
6770     caddr_t te;
6771     size_t tpgsz = page_get_pagesize(tszc);
6772
6773     ASSERT(svd->type == MAP_SHARED);
6774     ASSERT(flag == SDR_END);
6775     ASSERT(tszc < szc && tszc > 0);
6776
6777     if (eaddr > badseg1->s_base + badseg1->s_size) {
6778         te = badseg1->s_base + badseg1->s_size;
6779     } else {
6780         te = eaddr;
6781     }
6782
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, tszvvec);
6789             if (err != 0)
6790                 return (err);
6791         }
6792     }

```

```

6793                                         return (segvn_demote_range(badseg1, ta,
6794                                         te - ta, SDR_END, tszvvec));
6795                                         }
6796                                         }
6797                                         }
6798
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);
6808
6809     if (szc > 1 && (tszvvec = P2PHASE(szvvec, 1 << szc)) > 1) {
6810         uint_t tszc = highbit(tszvvec) - 1;
6811         size_t tpgsz = page_get_pagesize(tszc);
6812
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);
6819
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, tszvvec));
6824         }
6825     }
6826
6827     return (0);
6828 }
6829
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;
6835
6836     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6837
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;
6844
6845         err = ((svd->prot & prot) != prot) ? EACCES : 0;
6846         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
6847         return (err);
6848     }
6849
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);
6880             do {
6881                 pgno--;
6882                 protv[pgno] = VPP_PROT(&svd->vpage[pgno+pgoff]);
6883             } while (pgno != 0);
6884         }
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;
6895     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6896     return (svd->offset + (uintptr_t)(addr - seg->s_base));
6897 }

6901 /*ARGSUSED*/
6902 static int
6903 segvn_gettime(struct seg *seg, caddr_t addr)
6904 {
6905     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6906     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6907     return (svd->type | (svd->flags & (MAP_NORESERVE | MAP_TEXT |
6908         MAP_INITDATA)));
6909 }

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;
6918     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6919     *vpp = svd->vp;
6920     return (0);
6921 }

6922 }

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 *vpl, *vp2;
6943     u_offset_t off1, off2;
6944     struct anon_map *amp;

6945     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
6946     ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock) ||
6947            SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

6948     if (addr + delta < seg->s_base ||
6949         addr + delta >= (seg->s_base + seg->s_size))
6950         return (-1); /* exceeded segment bounds */

6951     pd = delta / (ssize_t)PAGESIZE; /* divide to preserve sign bit */
6952     page = seg_page(seg, addr);

6953 /*
6954  * Check to see if either of the pages addr or addr + delta
6955  * have advice set that prevents klustering (if MADV_RANDOM advice
6956  * is set for entire segment, or MADV_SEQUENTIAL is set and delta
6957  * is negative).
6958 */
6959 if (svd->advice == MADV_RANDOM ||
6960     svd->advice == MADV_SEQUENTIAL && delta < 0)
6961     return (-1);
6962 else if (svd->pageadvice && svd->vpage) {
6963     struct vpage *bvpp, *evpp;

6964     bvpp = &svd->vpage[page];
6965     evpp = &svd->vpage[page + pd];
6966     if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
6967         VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
6968         return (-1);
6969     if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
6970         VPP_ADVICE(evpp) == MADV_RANDOM)
6971         return (-1);
6972 }

6973 if (svd->type == MAP_SHARED)
6974     return (0); /* shared mapping - all ok */

6975 if ((amp = svd->amp) == NULL)
6976     return (0); /* off original vnode */

6977 page += svd->anon_index;

6978 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

6979 oap = anon_get_ptr(amp->ahp, page);
6980 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             }
7100             pp = page_lookup_nowait(vp, off, SE_SHARED);
7101             if (pp == NULL)
7102                 continue;

7103             /*
7104              * Examine the page to see whether it can be tossed out,
7105              * keeping track of how many we've found.
7106              */
7107             if (!page_tryupgrade(pp)) {
7108                 /*
7109                  * If the page has an i/o lock and no mappings,
7110                  * it's very likely that the page is being
7111                  * written out as a result of klustering.
7112                  * Assume this is so and take credit for it here.
7113                  */
7114                 if (!page_io_trylock(pp)) {
7115                     if (!hat_page_is_mapped(pp))
7116                         pgcnt++;
7117                 } else {
7118                     page_io_unlock(pp);
7119                 }
7120             }
7121         }
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_RELEASE(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 }

7277 vpp = svd->vpage;
7278 offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7279 bflags = ((flags & MS_ASYNC) ? B_ASYNC : 0) |
7280     ((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             }
7370             if (ap != NULL && pp->p_szc != 0 &&
7371                 page_tryupgrade(pp)) {
7372                 if (pp->p_lckcnt == 0 &&
7373                     pp->p_cowcnt == 0) {
7374                     /*
7375                     * swapfs VN_DISPOSE() won't
7376                     * invalidate large pages.
7377                     * Attempt to demote.
7378                     * XXX can't help it if it
7379                     * fails. But for swapfs
7380                     * pages it is no big deal.
7381                     */
7382                     (void) page_try_demote_pages(
7383                         pp);
7384                 }
7385             }
7386         }
7387         page_unlock(pp);
7388     }

```

```

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          */
7393
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         }
7408         /*
7409          * Note ISM pages are created large so (vp, off)'s
7410          * page cannot suddenly become large after we unlock
7411          * pp.
7412          */
7413         page_unlock(pp);
7414
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                       (bfflags | (IS_SWAPFSVP(vp) ? B_PAGE_NOWAIT : 0)),
7423                       svd->cred, NULL);
7424
7425     VN_REL(vp);
7426     if (err)
7427         break;
7428
7429     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7430
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));
7454
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     }
7461
7462     p = seg_page(seg, addr);
7463     ep = seg_page(seg, addr + len);
7464     start = svd->vp ? SEG_PAGE_VNODEBACKED : 0;
7465
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;
7486
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         }
7508
7509         /*
7510          * Gather vnode statistics
7511          */
7512         vp = svd->vp;
7513         offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7514
7515         if (vp != NULL) {
7516             /*
7517              * Try to obtain a "shared" lock on the page
7518              * without blocking. If this fails, determine
7519              * if the page is in memory.
7520              */

```

```

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 }

7537 /* Gather virtual page information */
7538 if (vpp) {
7539     if (VPP_ISPLOCK(vpp))
7540         ret |= SEG_PAGE_LOCKED;
7541     vpp++;
7542 }
7543
7544     *vec++ = (char)ret;
7545 }
7546 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7547 return (len);
7548 }

7549 */

7550 /* Statement for p_cowcnts/p_lckcnts.
7551 */
7552 /* p_cowcnt is updated while mlock/munlocking MAP_PRIVATE and PROT_WRITE region
7553 * irrespective of the following factors or anything else:
7554 */
7555 /* (1) anon slots are populated or not
7556 * (2) cow is broken or not
7557 * refcnt on ap is 1 or greater than 1
7558 */
7559 /* If it's not MAP_PRIVATE and PROT_WRITE, p_lckcnt is updated during mlock
7560 * and munlock.
7561 */
7562
7563
7564 /* Handling p_cowcnts/p_lckcnts during copy-on-write fault:
7565 */
7566 /* if vpage has PROT_WRITE
7567 * transfer cowcnt on the oldpage -> cowcnt on the newpage
7568 * else
7569 *     transfer lckcnt on the oldpage -> lckcnt on the newpage
7570 */
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     */

7591 /*
7592  * Lock down (or unlock) pages mapped by this segment.
7593  */
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;

7624 /*
7625  * Hold write lock on address space because may split or concatenate
7626  * segments
7627  */
7628 ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7629 /*
7630  * If this is a shm, use shm's project and zone, else use
7631  * project and zone of calling process
7632  */
7633
7634
7635 /* Determine if this segment backs a sysV shm */
7636 if (svd->amp != NULL && svd->amp->a_sp != NULL) {
7637     ASSERT(svd->type == MAP_SHARED);
7638     ASSERT(svd->tr_state == SEGVN_TR_OFF);
7639     sp = svd->amp->a_sp;
7640     proj = sp->shm_perm.ipc_proj;
7641     chargeproc = 0;
7642 }
7643
7644 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
7645 if (attr) {
7646     pageprot = attr & ~(SHARED|PRIVATE);
7647     segtype = attr & SHARED ? MAP_SHARED : MAP_PRIVATE;
7648 }
7649
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
7664     if (op == MC_LOCK) {
7665         if (svd->tr_state == SEGVN_TR_INIT) {
7666             svd->tr_state = SEGVN_TR_OFF;
7667         } else if (svd->tr_state == SEGVN_TR_ON) {
7668             ASSERT(svd->amp != NULL);
7669             segvn_textunrepr(seg, 0);
7670             ASSERT(svd->amp == NULL &&
7671                     svd->tr_state == SEGVN_TR_OFF);
7672         }
7673     }
7674
7675     /*
7676      * If we're locking, then we must create a vpage structure if
7677      * none exists.  If we're unlocking, then check to see if there
7678      * is a vpage -- if not, then we could not have locked anything.
7679     */
7680
7681     if ((vpp = svd->vpage) == NULL) {
7682         if (op == MC_LOCK)
7683             segvn_vpage(seg);
7684         else {
7685             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7686             return (0);
7687         }
7688     }
7689
7690     /*
7691      * The anonymous data vector (i.e., previously
7692      * unreferenced mapping to swap space) can be allocated
7693      * by lazily testing for its existence.
7694     */
7695     if (op == MC_LOCK && svd->amp == NULL && svd->vp == NULL) {
7696         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
7697         svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
7698         svd->amp->a_szc = seg->s_szc;
7699     }
7700
7701     if ((amp = svd->amp) != NULL) {
7702         anon_index = svd->anon_index + seg_page(seg, addr);
7703     }
7704
7705     offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7706     evp = &svd->vpage[seg_page(seg, addr + len)];
7707
7708     if (sp != NULL)
7709         mutex_enter(&sp->shm_mlock);
7710
7711     /* determine number of unlocked bytes in range for lock operation */
7712     if (op == MC_LOCK) {
7713
7714         if (sp == NULL) {
7715             for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;

```

```

7716                         vpp++) {
7717             if (!VPP_ISPPLOCK(vpp))
7718                 unlocked_bytes += PAGESIZE;
7719         }
7720     } else {
7721         ulong_t i_idx, i_edx;
7722         anon_sync_obj_t i_cookie;
7723         struct anon *i_ap;
7724         struct vnode *i_vp;
7725         u_offset_t i_off;
7726
7727         /* Only count sysV pages once for locked memory */
7728         i_edx = svd->anon_index + seg_page(seg, addr + len);
7729         ANON_LOCK_ENTER(&amp;->a_rwlock, RW_READER);
7730         for (i_idx = anon_index; i_idx < i_edx; i_idx++) {
7731             anon_array_enter(amp, i_idx, &i_cookie);
7732             i_ap = anon_get_ptr(amp->ahp, i_idx);
7733             if (i_ap == NULL) {
7734                 unlocked_bytes += PAGESIZE;
7735                 anon_array_exit(&i_cookie);
7736                 continue;
7737             }
7738             swap_xlate(i_ap, &i_vp, &i_off);
7739             anon_array_exit(&i_cookie);
7740             pp = page_lookup(i_vp, i_off, SE_SHARED);
7741             if (pp == NULL) {
7742                 unlocked_bytes += PAGESIZE;
7743                 continue;
7744             } else if (pp->p_lckcnt == 0)
7745                 unlocked_bytes += PAGESIZE;
7746             page_unlock(pp);
7747         }
7748         ANON_LOCK_EXIT(&amp;->a_rwlock);
7749     }
7750
7751     mutex_enter(&p->p_lock);
7752     err = rctl_incr_locked_mem(p, proj, unlocked_bytes,
7753                               chargeproc);
7754     mutex_exit(&p->p_lock);
7755
7756     if (err) {
7757         if (sp != NULL)
7758             mutex_exit(&sp->shm_mlock);
7759             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7760             return (err);
7761     }
7762
7763     /*
7764      * Loop over all pages in the range. Process if we're locking and
7765      * page has not already been locked in this mapping; or if we're
7766      * unlocking and the page has been locked.
7767     */
7768     for (vpp = &svd->vpage[seg_page(seg, addr)]; vpp < evp;
7769     vpp++, pos++, addr += PAGESIZE, offset += PAGESIZE, anon_index++) {
7770         if ((attr == 0 || VPP_PROT(vpp) == pageprot) &&
7771             ((op == MC_LOCK && !VPP_ISPPLOCK(vpp)) ||
7772              (op == MC_UNLOCK && VPP_ISPPLOCK(vpp)))) {
7773
7774             if (amp != NULL)
7775                 ANON_LOCK_ENTER(&amp;->a_rwlock, RW_READER);
7776
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);
7788
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             }
7807
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             }
7836
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;
7852
7853             ASSERT(vp != NULL);
7854
7855             error = VOP_GETPAGE(vp, (offset_t)off, PAGESIZE,
7856                               (uint_t *)NULL, pl, PAGESIZE, seg, addr,
7857                               S_OTHER, svd->cred, NULL);
7858
7859             if (error && ap != NULL) {
7860                 anon_array_exit(&cookie);
7861                 ANON_LOCK_EXIT(&amp->a_rwlock);
7862             }
7863
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             /*
7881             * Quit if we fail to fault in the page. Treat
7882             * the failure as an error, unless the addr
7883             * is mapped beyond the end of a file.
7884             */
7885             if (error && svd->vp) {
7886                 va.va_mask = AT_SIZE;
7887                 if (VOP_GETATTR(svd->vp, &va, 0,
7888                               svd->cred, NULL) != 0) {
7889                     err = EIO;
7890                     goto out;
7891                 }
7892                 if (btopr(va.va_size) >=
7893                     btopr(off + 1)) {
7894                     err = EIO;
7895                     goto out;
7896                 }
7897             }
7898             /*
7899             * else if (error) {
7900                 err = EIO;
7901                 goto out;
7902             }
7903             pp = pl[0];
7904             ASSERT(pp != NULL);
7905
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             */
7917
7918             if (pl[0] != NULL) {
7919                 if (pl[0] <= off) {
7920                     if (pl[0] > off) {
7921                         if (pl[0] - off >= PAGESIZE) {
7922                             if (pl[0] - off >= PAGESIZE) {
7923                                 if (pl[0] - off >= PAGESIZE) {
7924                                     if (pl[0] - off >= PAGESIZE) {
7925                                         if (pl[0] - off >= PAGESIZE) {
7926                                             if (pl[0] - off >= PAGESIZE) {
7927                                                 if (pl[0] - off >= PAGESIZE) {
7928                                                     if (pl[0] - off >= PAGESIZE) {
7929                                                         if (pl[0] - off >= PAGESIZE) {
7930                                                             if (pl[0] - off >= PAGESIZE) {
7931                                                                 if (pl[0] - off >= PAGESIZE) {
7932                                                                     if (pl[0] - off >= PAGESIZE) {
7933                                                                         if (pl[0] - off >= PAGESIZE) {
7934                                                                             if (pl[0] - off >= PAGESIZE) {
7935                                                                                 if (pl[0] - off >= PAGESIZE) {
7936                                                                 if (pl[0] - off >= PAGESIZE) {
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
7950
7951
7952
7953
7954
7955
7956
7957
7958
7959
7960
7961
7962
7963
7964
7965
7966
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
8000
8001
8002
8003
8004
8005
8006
8007
8008
8009
8010
8011
8012
8013
8014
8015
8016
8017
8018
8019
8020
8021
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
8050
8051
8052
8053
8054
8055
8056
8057
8058
8059
8060
8061
8062
8063
8064
8065
8066
8067
8068
8069
8070
8071
8072
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
8100
8101
8102
8103
8104
8105
8106
8107
8108
8109
8110
8111
8112
8113
8114
8115
8116
8117
8118
8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
8150
8151
8152
8153
8154
8155
8156
8157
8158
8159
8160
8161
8162
8163
8164
8165
8166
8167
8168
8169
8170
8171
8172
8173
8174
8175
8176
8177
8178
8179
8180
8181
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
8200
8201
8202
8203
8204
8205
8206
8207
8208
8209
8210
8211
8212
8213
8214
8215
8216
8217
8218
8219
8220
8221
8222
8223
8224
8225
8226
8227
8228
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
8250
8251
8252
8253
8254
8255
8256
8257
8258
8259
8260
8261
8262
8263
8264
8265
8266
8267
8268
8269
8270
8271
8272
8273
8274
8275
8276
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
8300
8301
8302
8303
8304
8305
8306
8307
8308
8309
8310
8311
8312
8313
8314
8315
8316
8317
8318
8319
8320
8321
8322
8323
8324
8325
8326
8327
8328
8329
8330
8331
8332
8333
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
8350
8351
8352
8353
8354
8355
8356
8357
8358
8359
8360
8361
8362
8363
8364
8365
8366
8367
8368
8369
8370
8371
8372
8373
8374
8375
8376
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
8400
8401
8402
8403
8404
8405
8406
8407
8408
8409
8410
8411
8412
8413
8414
8415
8416
8417
8418
8419
8420
8421
8422
8423
8424
8425
8426
8427
8428
8429
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
8450
8451
8452
8453
8454
8455
8456
8457
8458
8459
8460
8461
8462
8463
8464
8465
8466
8467
8468
8469
8470
8471
8472
8473
8474
8475
8476
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
8500
8501
8502
8503
8504
8505
8506
8507
8508
8509
8510
8511
8512
8513
8514
8515
8516
8517
8518
8519
8520
8521
8522
8523
8524
8525
8526
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
8550
8551
8552
8553
8554
8555
8556
8557
8558
8559
8560
8561
8562
8563
8564
8565
8566
8567
8568
8569
8570
8571
8572
8573
8574
8575
8576
8577
8578
8579
8580
8581
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
8600
8601
8602
8603
8604
8605
8606
8607
8608
8609
8610
8611
8612
8613
8614
8615
8616
8617
8618
8619
8620
8621
8622
8623
8624
8625
8626
8627
8628
8629
8630
8631
8632
8633
8634
8635
8636
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
8650
8651
8652
8653
8654
8655
8656
8657
8658
8659
8660
8661
8662
8663
8664
8665
8666
8667
8668
8669
8670
8671
8672
8673
8674
8675
8676
8677
8678
8679
8680
8681
8682
8683
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
8700
8701
8702
8703
8704
8705
8706
8707
8708
8709
8710
8711
8712
8713
8714
8715
8716
8717
8718
8719
8720
8721
8722
8723
8724
8725
8726
8727
8728
8729
8730
8731
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
8750
8751
8752
8753
8754
8755
8756
8757
8758
8759
8760
8761
8762
8763
8764
8765
8766
8767
8768
8769
8770
8771
8772
8773
8774
8775
8776
8777
8778
8779
8780
8781
8782
8783
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
8800
8801
8802
8803
8804
8805
8806
8807
8808
8809
8810
8811
8812
8813
8814
8815
8816
8817
8818
8819
8820
8821
8822
8823
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
8850
8851
8852
8853
8854
8855
8856
8857
8858
8859
8860
8861
8862
8863
8864
8865
8866
8867
8868
8869
8870
8871
8872
8873
8874
8875
8876
8877
8878
8879
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
8900
8901
8902
8903
8904
8905
8906
8907
8908
8909
8910
8911
8912
8913
8914
8915
8916
8917
8918
8919
8920
8921
8922
8923
8924
8925
8926
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
8950
8951
8952
8953
8954
8955
8956
8957
8958
8959
8960
8961
8962
8963
8964
8965
8966
8967
8968
8969
8970
8971
8972
8973
8974
8975
8976
8977
8978
8979
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
9000
9001
9002
9003
9004
9005
9006
9007
9008
9009
9010
9011
9012
9013
9014
9015
9016
9017
9018
9019
9020
9021
9022
9023
9024
9025
9026
9027
9028
9029
9030
9031
9032
9033
9034
9035
9036
9037
9038
9039
9040
9041
9042
9043
9044
9045
9046
9047
9048
9049
9050
9051
9052
9053
9054
9055
9056
9057
9058
9059
9060
9061
9062
9063
9064
9065
9066
9067
9068
9069
9070
9071
9072
9073
9074
9075
9076
9077
9078
9079
9080
9081
9082
9083
9084
9085
9086
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
9100
9101
9102
9103
9104
9105
9106
9107
9108
9109
9110
9111
9112
9113
9114
9115
9116
9117
9118
9119
9120
9121
9122
9123
9124
9125
9126
9127
9128
9129
9130
9131
9132
9133
9134
9135
9136
9137
9138
9139
9140
9141
9142
9143
9144
9145
9146
9147
9148
9149
9150
9151
9152
9153
9154
9155
9156
9157
9158
9159
9160
9161
9162
9163
9164
9165
9166
9167
9168
9169
9170
9171
9172
9173
9174
9175
9176
9177
9178
9179
9180
9181
9182
9183
9184
9185
9186
9187
9188
9189
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
9200
9201
9202
9203
9204
9205
9206
9207
9208
9209
9210
9211
9212
9213
9214
9215
9216
9217
9218
9219
9220
9221
9222
9223
9224
9225
9226
9227
9228
9229
9230
9231
9232
9233
9234
9235
9236
9237
9238
9239
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249
9250
9251
9252
9253
9254
9255
9256
9257
9258
9259
9260
9261
9262
9263
9264
9265
9266
9267
9268
9269
9270
9271
9272
9273
9274
9275
9276
9277
9278
9279
9280
9281
9282
9283
9284
9285
9286
9287
9288
9289
9290
9291
9292
9293
9294
9295
9296
9297
9298
9299
9300
9301
9302
9303
9304
9305
9306
9307
9308
9309
9310
9311
9312
9313
9314
9315
9316
9317
9318
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
9350
9351
9352
9353
9354
9355
9356
9357
9358
9359
9360
9361
9362
9363
9364
9365
9366
9367
9368
9369
9370
9371
9372
9373
9374
9375
9376
9377
9378
9379
9380
9381
9382
9383
9384
9385
9386
9387
9388
9389
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
9400
9401
9402
9403
9404
9405
9406
9407
9408
9409
9410
9411
9412
9413
9414
9415
9416
9417
9418
9419
9420
9421
9422
9423
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
9450
9451
9452
9453
9454
9455
9456
9457
9458
9459
9460
9461
9462
9463
9464
9465
9466
9467
9468
9469
9470
9471
9472
9473
9474
9475
9476
9477
9478
9479
9480
9481
9482
9483
9484
9485
9486
9487
9488
9489
9490
9491
9492
9493
9494
9495
9496
9497
9498
9499
9500
9501
9502
9503
9504
9505
9506
9507
9508
9509
9510
9511
9512
9513
9514
9515
9516
9517
9518
9519
9520
9521
9522
9523
9524
9525
9526
9527
9528
9529
9530
9531
9532
9533
9534
9535
9536
9537
9538
9539
9540
9541
9542
9543
9544
9545
9546
9547
9548
9549
9550
9551
9552
9553
9554
9555
9556
9557
9558
9559
9560
9561
9562
9563
9564
9565
9566
9567
9568
9569
9570
9571
9572
9573
9574
9575
9576
9577
9578
9579
9580
9581
9582
9583
9584
9585
9586
9587
9588
9589
9590
9591
9592
9593
9594
9595
9596
9597
9598
9599
9600
9601
9602
9603
9604
9605
9606
9607
9608
9609
9610
9611
9612
9613
9614
9615
9616
9617
9618
9619
9620
9621
9622
9623
9624
9625
9626
9627
9628
9629
9630
9631
9632
9633
9634
9635
9636
9637
9638
9639
9640
9641
9642
9643
9644
9645
9646
9647
9648
9649
9650
9651
9652
9653
9654
9655
9656
9657
9658
9659
9660
9661
9662
9663
9664
9665
9666
9667
9668
9669
9670
9671
9672
9673
9674
9675
9676
9677
9678
9679
9680
9681
9682
9683
9684
9685
9686
9687
9688
9689
9690
9691
9692
9693
9694
9695
9696
9697
9698
9699
9700
9701
9702
9703
9704
9705
9706
9707
9708
9709
9710
9711
9712
9713
9714
9715
9716
9717
9718
9719
9720
9721
9722
9723
9724
9725
9726
9727
9728
9729
9730
9731
9732
9733
9734
9735
9736
9737
9738
9739
9740
9741
9742
9743
9744
9745
9746
9747
9748
9749
9750
9751
9752
9753
9754
9755
9756
9757
9758
9759
9760
9761
9762
9763
9764

```

```

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(&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     } else {
7992         /* Account bytes that were unlocked */
7993         if (unlocked_bytes > 0) {
7994             if (proj == NULL)
7995                 mutex_enter(&p->p_lock);
7996             rctl_decr_locked_mem(p, proj, unlocked_bytes,
7997                 chargeproc);
7998             if (proj == NULL)
7999                 mutex_exit(&p->p_lock);
8000         }
8001     }
8002     if (sp != NULL)
8003         mutex_exit(&sp->shm_mlock);
8004     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8005
8006     return (err);
8007 }
8008
8009 /*
8010  * Set advice from user for specified pages
8011  * There are 5 types of advice:
8012  *      MADV_NORMAL      - Normal (default) behavior (whatever that is)
8013  *      MADV_RANDOM      - Random page references
8014  *                      do not allow readahead or 'klustering'
8015  *      MADV_SEQUENTIAL   - Sequential page references
8016  *                      Pages previous to the one currently being
8017  *                      accessed (determined by fault) are 'not needed'
8018  *                      and are freed immediately
8019  *      MADV_WILLNEED    - Pages are likely to be used (fault ahead in mctl)
8020  *      MADV_DONTNEED    - Pages are not needed (synced out in mctl)
8021  *      MADV_FREE        - Contents can be discarded
8022  *      MADV_ACCESS_DEFAULT - Default access
8023  *      MADV_ACCESS_LWP  - Next LWP will access heavily
8024  *      MADV_ACCESS_MANY - Many LWP's or processes will access heavily
8025  */
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     }
8056
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     */
8064
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     }
8072
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
8120 amp = svd->amp;
8121 vp = svd->vp;
8122 if (behav == MADV_FREE) {
8123     /*
8124      * MADV_FREE is not supported for segments with
8125      * underlying object; if anonmap is NULL, anon slots
8126      * are not yet populated and there is nothing for
8127      * us to do. As MADV_FREE is advisory, we don't
8128      * return error in either case.
8129     */
8130 if (vp != NULL || amp == NULL) {
8131     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8132     return (0);
8133 }
8134
8135 segvn_purge(seg);
8136
8137 page = seg_page(seg, addr);
8138 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
8139 anon_disclaim(amp, svd->anon_index + page, len);
8140 ANON_LOCK_EXIT(&amp->a_rwlock);
8141 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8142 return (0);
8143
8144 }
8145
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     }
8174     already_set = lgrp_privm_policy_set(policy,
8175                                           &svd->policy_info, len);
8176 }
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_ctrl() 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
8230 case MADV_SEQUENTIAL:
8231     /*
8232      * unloading mapping guarantees
8233      * detection in segvn_fault
8234      */
8235     ASSERT(seg->s_szc == 0);
8236     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
8237     hat_unload(seg->s_as->a_hat, addr, len,
8238                HAT_UNLOAD);
8239
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     }
8320
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);
8327
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;
8335
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     }
8344
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);
8359
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;
8367
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);
8380
8381     next = AS_SEGNEXT(new_seg->s_as,
8382                       new_seg);
8383
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
8391
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);
8404
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);
8415
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;
8430
8431         (void) lgrp_privm_policy_set(policy,
8432                 &svd->policy_info, seg->s_size);
8433
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);

```

```

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_mtblf = 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 **plist, **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_cbfnc_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;

8656 #ifdef DEBUG
8657     if (type == L_PAGELOCK && segvn_pglock_mtblf) {
8658         hrtim_t ts = gethrtime();
8659         if ((ts % segvn_pglock_mtblf) == 0) {
8660             return (ENOTSUP);
8661         }
8662         if ((ts % segvn_pglock_mtblf) == 1) {
8663             return (EFAULT);
8664         }
8665     }
8666 #endif
8667 TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
8668         "segvn_pagelock: start seg %p addr %p", seg, addr);
8669 ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
8670 ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);
8671 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
8672 /*
8673 * for now we only support pagelock to anon memory. We would have to
8674 * check protections for vnode objects and call into the vnode driver.
8675 * That's too much for a fast path. Let the fault entry point handle
8676 * it.
8677 */
8678 if (svd->vp != NULL) {
8679     if (type == L_PAGELOCK) {
8680         error = ENOTSUP;
8681         goto out;
8682     }
8683     panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
8684 }
8685 if ((amp = svd->amp) == NULL) {
8686     if (type == L_PAGELOCK) {
8687         error = EFAULT;
8688         goto out;
8689     }
8690     panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
8691 }
8692 if (rw != S_READ && rw != S_WRITE) {
8693     if (type == L_PAGELOCK) {
8694         error = ENOTSUP;
8695         goto out;
8696     }
8697     panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
8698 }
8699 if (seg->s_zc != 0) {
8700     /* */
8701 }
8702 }
8703 */
8704 if (seg->s_zc != 0) {
8705     /* */
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_pglock() 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_pglock() 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     lpgeaddr = addr + len;
8800     if (pflags) {
8801         if (svd->type == MAP_PRIVATE) {
8802             lpgeaddr = (caddr_t)P2ROUNDUP(
8803                 (uintptr_t)lpgeaddr,
8804                 segvn_pglock_comb_balign);
8805         } else {
8806             ulong_t aix = svd->anon_index +
8807                 seg_page(seg, lpgeaddr);
8808             ulong_t aaix = P2ROUNDUP(aix,
8809                 segvn_pglock_comb_palign);
8810             if (aaix < aix) {
8811                 lpgeaddr = 0;
8812             } else {
8813                 lpgeaddr += ptob(aaix - aix);
8814             }
8815         }
8816     }
8817     if (lpgeaddr == 0 || lpgeaddr > seg->s_base + seg->s_size) {
8818         lpgeaddr = seg->s_base + seg->s_size;
8819         sftlck_send = 1;
8820     }
8821 }
8822 if (svd->pageprot && lpgeaddr != addr + len) {
8823     struct vpage *vp;
8824     struct vpage *evp;
8825     vp = &svd->vpage[seg_page(seg, addr + len)];
8826     evp = &svd->vpage[seg_page(seg, lpgeaddr)];
8827     while (vp < evp) {
8828         if ((VPP_PROT(vp) & protchk) == 0) {
8829             break;
8830         }
8831         vp++;
8832     }
8833     if (vp < evp) {
8834         lpgeaddr = addr + len;
8835     }
8836     if (vp < evp) {
8837         lpgeaddr = addr + len;
8838     }

```

```

8839         }
8840         adjustpages = btop((uintptr_t)(addr - lpgaddr));
8841     }
8842
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         proclaim_callback = shamp_reclaim;
8854     } else {
8855         pamp = NULL;
8856         paddr = lpgaddr;
8857         proclaim_callback = segvn_reclaim;
8858     }
8859
8860     if (type == L_PAGEUNLOCK) {
8861         VM_STAT_ADD(segvnmstats.pagelock[0]);
8862
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     }
8880
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         (*proclaim_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, proclaim_callback);
8909     }
8910
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     }
8916
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     }
8925
8926     /*
8927      * If someone is blocked while unmapping, we purge
8928      * segment page cache and thus reclaim plist 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              * unmappwait 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 }
8964
8965 /* The L_PAGELOCK case ... */
8966 VM_STAT_ADD(segvnmstats.pagelock[1]);
8967
8968 /*
8969  * 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;
8986
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;
8996
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     }
9007
9008     /*
9009     * try to find pages in segment page cache
9010     */
9011     plist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
9012     if (plist != 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 = plist + 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     }
9031
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;
9056
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     }
9075
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     plist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
9104     pl = plist;
9105     *ppp = plist + 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;
9113
9114     page = seg_page(seg, addr);
9115     anon_index = svd->anon_index + page;
9116
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;
9125
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);
9140
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;
9152
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_vp != 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     *plist++ = pp;
9198
9199     if (anlock) {
9200         /* Ensure the lock is dropped */
9201         anon_array_exit(&cookie);
9202     }
9203     ANON_LOCK_EXIT(&amp;->a_rwlock);
9204
9205     if (a >= addr + len) {
9206         atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
9207         if (pamp != NULL) {
9208             ASSERT(svd->type == MAP_SHARED);
9209             atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
9210                           npages);
9211             wlen = len;
9212         }
9213         if (sftlck_sbase) {
9214             atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9215         }
9216         if (sftlck_send) {
9217             atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9218         }
9219         if (use_pcache) {
9220             (void) seg_pinsert(seg, pamp, paddr, len, wlen, pl,
9221                               rw, pflags, preclaim_callback);
9222         }
9223         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9224         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
9225                 "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
9226         return (0);
9227     }
9228
9229     plist = pl;
9230     np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
9231     while (np > (uint_t)0) {
9232         ASSERT(PAGE_LOCKED(*plist));
9233         page_unlock(*plist);
9234         np--;
9235         plist++;

```

```

9235     }
9236     kmem_free(pl, sizeof (page_t *) * (npages + 1));
9237 out:    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9238     *ppp = NULL;
9239     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
9240             "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
9241     return (error);
9242 }
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;
9252
9253     /*
9254      * pcache is only used by pure anon segments.
9255      */
9256     if (svd->amp == NULL || svd->vp != NULL) {
9257         return;
9258     }
9259
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 */
9278 /*ARGSUSED*/
9279 static int
9280 segvn_reclaim(void *ptag, caddr_t addr, size_t len, struct page **plist,
9281                 enum seg_rw rw, int async)
9282 {
9283     struct seg *seg = (struct seg *)ptag;
9284     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9285     pgcnt_t np, npages;
9286     struct page **pl;
9287
9288     npages = np = btop(len);
9289     ASSERT(npages);
9290
9291     ASSERT(svd->vp == NULL && svd->amp != NULL);
9292     ASSERT(svd->softlockcnt >= npages);
9293     ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9294
9295     pl = plist;
9296
9297     ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
9298     ASSERT(!async || pl[np] == PCACHE_SHWLIST);

```

```

9301     while (np > (uint_t)0) {
9302         if (rw == S_WRITE) {
9303             hat_setrefmod(*plist);
9304         } else {
9305             hat_setref(*plist);
9306         }
9307         page_unlock(*plist);
9308         np--;
9309         plist++;
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 decrement 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 nouunmapwait 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 shmp_reclaim(void *ptag, caddr_t addr, size_t len, struct page **plist,
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 = plist;
9371     ASSERT(pl[np] == NOPCACHE_SHWLIST || pl[np] == PCACHE_SHWLIST);
9372     ASSERT(!async || pl[np] == PCACHE_SHWLIST);
9374
9375     while (np > (uint_t)0) {
9376         if (rw == S_WRITE) {
9377             hat_setrefmod(*plist);
9378         } else {
9379             hat_setref(*plist);
9380         }
9381         page_unlock(*plist);
9382         np--;
9383         plist++;
9385     }
9386     kmem_free(pl, sizeof (page_t *) * (npages + 1));
9387
9388     /*
9389      * If somebody sleeps in anonmap_purge() wake them up if a_softlockcnt
9390      * drops to 0. anon map can't be freed until a_softlockcnt drops to 0
9391      * and anonmap_purge() acquires a_purgemtx.
9392
9393     mutex_enter(&amp;->a_purgemtx);
9394     if (!atomic_add_long_nv((ulong_t *)&amp;->a_softlockcnt, -npages) &&
9395         amp->a_purgewait) {
9396         amp->a_purgewait = 0;
9397         cv_broadcast(&amp;->a_purgecv);
9398     }
9399     mutex_exit(&amp;->a_purgemtx);
9400     return (0);
9401
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_map *amp = NULL;
9413     ulong_t anon_index;
9414     struct anon_map *amp;
9415     anon_sync_obj_t cookie;
9416
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     }
9422
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 {
9430             SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
9431             if ((amp = ssvd->amp) != NULL) {
9432

```

```

9433             anon_index = ssvd->anon_index +
9434                 seg_page(seg, addr);
9435         }
9436         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9437
9438         ASSERT(amp != NULL);
9439
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;
9445             pp = anon_zero(seg, addr, &ap, ssvd->cred);
9446             if (pp == NULL) {
9447                 anon_array_exit(&cookie);
9448                 ANON_LOCK_EXIT(&amp;->a_rwlock);
9449                 return (ENOMEM);
9450             }
9451             ASSERT(anon_get_ptr(amp->ahp, anon_index)
9452                   == NULL);
9453             (void) anon_set_ptr(amp->ahp, anon_index,
9454                                 ap, ANON_SLEEP);
9455             page_unlock(pp);
9456         }
9457
9458         anon_array_exit(&cookie);
9459         ANON_LOCK_EXIT(&amp;->a_rwlock);
9460
9461         memidp->val[0] = (uintptr_t)ap;
9462         memidp->val[1] = (uintptr_t)addr & PAGEOFFSET;
9463
9464     }
9465
9466     return (EINVAL);
9467 }
9468
9469 static int
9470 sameprot(struct seg *seg, caddr_t a, size_t len)
9471 {
9472     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
9473     struct vpage *vpage;
9474     spgcnt_t pages = btop(len);
9475     uint_t prot;
9476
9477     if (svd->pageprot == 0)
9478         return (1);
9479
9480     ASSERT(svd->vpage != NULL);
9481
9482     vpage = &svd->vpage[seg_page(seg, a)];
9483     prot = VPP_PROT(vpage);
9484     vpage++;
9485     pages--;
9486     while (pages-- > 0) {
9487         if (prot != VPP_PROT(vpage))
9488             return (0);
9489         vpage++;
9490     }
9491     return (1);
9492 }
9493
9494 /*
9495  * Get memory allocation policy info for specified address in given segment
9496  */
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
9509     ASSERT(seg != NULL);
9510
9511     svn_data = (struct segvn_data *)seg->s_data;
9512     if (svn_data == NULL)
9513         return (NULL);
9514
9515     /*
9516      * Get policy info for private or shared memory
9517      */
9518     if (svn_data->type != MAP_SHARED) {
9519         if (svn_data->tr_state != SEGVN_TR_ON) {
9520             policy_info = &svn_data->policy_info;
9521         } else {
9522             policy_info = &svn_data->tr_policy_info;
9523             ASSERT(policy_info->mem_policy ==
9524                   LGRP_MEM_POLICY_NEXT_SEG);
9525         }
9526     } else {
9527         amp = svn_data->amp;
9528         anon_index = svn_data->anon_index + seg_page(seg, addr);
9529         vp = svn_data->vp;
9530         vn_off = svn_data->offset + (uintptr_t)(addr - seg->s_base);
9531         policy_info = lgrp_shm_policy_get(amp, anon_index, vp, vn_off);
9532     }
9533
9534     return (policy_info);
9535 }
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

```

```

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
9573     ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9574     ASSERT(SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9575     ASSERT(p != NULL);
9576     ASSERT(svd->tr_state == SEGVN_TR_INIT);
9577     ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9578     ASSERT(svd->flags & MAP_TEXT);
9579     ASSERT(svd->type == MAP_PRIVATE);
9580     ASSERT(vp != NULL && svd->amp == NULL);
9581     ASSERT(!svd->pageprot & !(svd->prot & PROT_WRITE));
9582     ASSERT(!(svd->flags & MAP_NORESERVE) && svd->swresv == 0);
9583     ASSERT(seg->s_as != &kas);
9584     ASSERT(off < eoff);
9585     ASSERT(svntr_hashtab != NULL);
9586
9587     /*
9588      * If numa optimizations are no longer desired bail out.
9589      */
9590     if (!lgrp_optimizations()) {
9591         svd->tr_state = SEGVN_TR_OFF;
9592         return;
9593     }
9594
9595     /*
9596      * Avoid creating anon maps with size bigger than the file size.
9597      * If VOP_GETATTR() call fails bail out.
9598      */
9599     va.va_mask = AT_SIZE | AT_MTIME | AT_CTIME;
9600     if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL) != 0) {
9601         svd->tr_state = SEGVN_TR_OFF;
9602         SEGVN_TR_ADDSTAT(gaerr);
9603         return;
9604     }
9605     if (btopr(va.va_size) < btopr(eoff)) {
9606         svd->tr_state = SEGVN_TR_OFF;
9607         SEGVN_TR_ADDSTAT(overmap);
9608         return;
9609     }
9610
9611     /*
9612      * VVMEXEC may not be set yet if exec() prefaults text segment. Set
9613      * this flag now before vn_is_mapped(V_WRITE) so that MAP_SHARED
9614      * mapping that checks if trache for this vnode needs to be
9615      * invalidated can't miss us.
9616      */
9617     if (!(vp->v_flag & VVMEXEC)) {
9618         mutex_enter(&vp->v_lock);
9619         vp->v_flag |= VVMEXEC;
9620         mutex_exit(&vp->v_lock);
9621     }
9622     mutex_enter(&svntr_hashtab[hash].tr_lock);
9623
9624     /*
9625      * Bail out if potentially MAP_SHARED writable mappings exist to this
9626      * vnode. We don't want to use old file contents from existing
9627      * replicas if this mapping was established after the original file
9628      * was changed.
9629      */
9630     if (vn_is_mapped(vp, V_WRITE)) {
9631         mutex_exit(&svntr_hashtab[hash].tr_lock);
9632         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     }
9640
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 modifying
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_lgrp_id;
9723 if (lgrp_id == LGRP_NONE) {
9724 /*
9725  * In case exec() prefaults text on non main thread use
9726  * current thread lgrp_id. It will become main thread anyway
9727  * soon.
9728 */
9729 lgrp_id = lgrp_home_id(curthread);
9730 }
9731 /*
9732  * Set p_tr_lgrp_id to lgrp_id 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_lgrp_id 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 asynchronous segvn thread rechecks if
9737  * segments should change lgrps used for text replication. If we fail
9738  * to set p_tr_lgrp_id 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_lgrp_id to NLGRPS_MAX at this point.
9743 */
9744 ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9745 olid = p->p_tr_lgrp_id;
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_lgrp_id, olid, nlid) !=
9749         olid) {
9750         olid = p->p_tr_lgrp_id;
9751         ASSERT(olid != LGRP_NONE);
9752         if (olid != lgrp_id && olid != NLGRPS_MAX) {
9753             p->p_tr_lgrp_id = NLGRPS_MAX;
9754         }
9755     }
9756     ASSERT(p->p_tr_lgrp_id != LGRP_NONE);
9757     membar_producer();
9758 */
9759 /*
9760  * lgrp_move_thread() won't schedule async recheck after
9761  * p->p_t1_lgrp_id update unless p->p_tr_lgrp_id is not
9762  * LGRP_NONE. Recheck p_t1_lgrp_id once now that p->p_tr_lgrp_id
9763  * 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 /* If no amp was created yet for lgrp_id create a new one as long as
9771 * we have enough memory to afford it.
9772 */
9773 if ((amp = svntr->tr_amp[lgrp_id]) == NULL) {
9774     size_t trmem = atomic_add_long_nv(&segvn_textrepl_bytes, size);
9775     if (trmem > segvn_textrepl_max_bytes) {
9776         SEGVN_TR_ADDSTAT(normem);
9777         goto fail;
9778     }
9779 if (anon_try_resv_zone(size, NULL) == 0) {
9780     SEGVN_TR_ADDSTAT(noanon);
9781     goto fail;
9782 }
9783 amp = anonmap_alloc(size, size, ANON_NOSLEEP);
9784 if (amp == NULL) {
9785     anon_unresv_zone(size, NULL);
9786     SEGVN_TR_ADDSTAT(nokmem);
9787     goto fail;
9788 }
9789 ASSERT(amp->refcnt == 1);
9790 amp->a_szc = szc;
9791 svntr->tr_amp[lgrp_id] = amp;
9792 SEGVN_TR_ADDSTAT(newamp);
9793 }
9794 svntr->tr_refcnt++;
9795 ASSERT(svd->svn_trnext == NULL);
9796 ASSERT(svd->svn_trprev == NULL);
9797 svd->svn_trnext = svntr->tr_svthead;
9798 svd->svn_trprev = NULL;
9799 if (svntr->tr_svthead != NULL) {
9800     svntr->tr_svthead->svn_trprev = svd;
9801 }
9802 svntr->tr_svthead = svd;
9803 ASSERT(amp->a_szc == szc && amp->size == size && amp->swresv == size);
9804 ASSERT(amp->refcnt >= 1);
9805 svd->amp = amp;
9806 svd->anon_index = 0;
9807 svd->tr_policy_info.mem_policy = LGRP_MEM_POLICY_NEXT_SEG;
9808 svd->tr_policy_info.mem_lgrpid = lgrp_id;
9809 svd->tr_state = SEGVN_TR_ON;
9810 mutex_exit(&svntr_hashtab[hash].tr_lock);
9811 SEGVN_TR_ADDSTAT(repl);
9812 return;
9813 fail:
9814     ASSERT(segvn_textrepl_bytes >= size);
9815     atomic_add_long(&segvn_textrepl_bytes, -size);
9816     ASSERT(svntr != NULL);
9817     ASSERT(svntr->tr_amp[lgrp_id] == NULL);
9818     if (svntr->tr_refcnt == 0) {
9819         ASSERT(svntr == svntr_hashtab[hash].tr_head);
9820         svntr_hashtab[hash].tr_head = svntr->tr_next;
9821         mutex_exit(&svntr_hashtab[hash].tr_lock);
9822         kmem_cache_free(svntr_cache, svntr);
9823     } else {
9824         mutex_exit(&svntr_hashtab[hash].tr_lock);
9825     }
9826 svd->tr_state = SEGVN_TR_OFF;
9827 }
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 **prv_svntrp;
9849     lgrp_id_t lgrp_id = svd->tr_policy_info.mem_lgrpid;
9850     lgrp_id_t i;
9851
9852     ASSERT(AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
9853     ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock)) ||
9854         SEGVN_WRITE_HELD(seg->s_as, &svd->lock));
9855     ASSERT(svd->tr_state == SEGVN_TR_ON);
9856     ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
9857     ASSERT(svd->amp != NULL);
9858     ASSERT(svd->amp->refcnt >= 1);
9859     ASSERT(svd->anon_index == 0);
9860     ASSERT(lgrp_id != LGRP_NONE && lgrp_id < NLGRPS_MAX);
9861     ASSERT(svntr_hashtab != NULL);
9862
9863     mutex_enter(&svntr_hashtab[hash].tr_lock);
9864     prv_svntrp = &svntr_hashtab[hash].tr_head;
9865     for (; (svntrp = *prv_svntrp) != NULL; prv_svntrp = &svntrp->tr_next) {
9866         ASSERT(svntrp->tr_refcnt != 0);
9867         if (svntrp->tr_vp == vp && svntrp->tr_off == off &&
9868             svntrp->tr_eoff == eoff && svntrp->tr_szc == szc) {
9869             break;
9870         }
9871     }
9872     if (svntrp == NULL) {
9873         panic("segvn_textunrepl: svntr record not found");
9874     }
9875     if (svntrp->tr_amp[lgrp_id] != svd->amp) {
9876         panic("segvn_textunrepl: amp mismatch");
9877     }
9878     svd->tr_state = SEGVN_TR_OFF;
9879     svd->amp = NULL;
9880     if (svd->svn_trprev == NULL) {
9881         ASSERT(svntrp->tr_svthead == svd);
9882         svntrp->tr_svthead = svd->svn_trnext;
9883         if (svntrp->tr_svthead != NULL) {
9884             svntrp->tr_svthead->svn_trprev = NULL;
9885         }
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 }
```

```

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;
9939
9940     ASSERT(vp->v_flag & VVMEEXEC);
9941
9942     if (svntr_hashtab == NULL) {
9943         return;
9944     }
9945
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 }
9956 static void
9957 segvn_trasync_thread(void)
9958 {
9959     callb_cpr_t cpr_info;

```

```

9961     kmutex_t cpr_lock;      /* just for CPR stuff */
9962
9963     mutex_init(&cpr_lock, NULL, MUTEX_DEFAULT, NULL);
9964
9965     CALLB_CPR_INIT(&cpr_info, &cpr_lock,
9966                     callb_generic_cpr, "segvn_async");
9967
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);
9975
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 }

9987 static uint64_t segvn_lgrp_trthr_migrs_snpsht = 0;
9988
9989 static void
9990 segvn_trupdate_wakeup(void *dummy)
9991 {
9992     uint64_t cur_lgrp_trthr_migrs = lgrp_get_trthr_migrations();
9993
9994     if (cur_lgrp_trthr_migrs != segvn_lgrp_trthr_migrs_snpsht) {
9995         segvn_lgrp_trthr_migrs_snpsht = cur_lgrp_trthr_migrs;
9996         sema_v(&segvn_trasync_sem);
9997     }
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 }

10006 static void
10007 segvn_trupdate(void)
10008 {
10009     ulong_t          hash;
10010     svntr_t          *svntrp;
10011     segvn_data_t     *svd;
10012
10013     ASSERT(svntr_hashtab != NULL);
10014
10015     for (hash = 0; hash < svntr_hashtab_sz; hash++) {
10016         mutex_enter(&svntr_hashtab[hash].tr_lock);
10017         svntrp = svntr_hashtab[hash].tr_head;
10018         for (; svntrp != NULL; svntrp = svntrp->tr_next) {
10019             ASSERT(svntrp->tr_refcnt != 0);
10020             svd = svntrp->tr_svnhead;
10021             for (; svd != NULL; svd = svd->svn_trnext) {
10022                 segvn_trupdate_seg(svd->seg, svd, svntrp,
10023                                     hash);
10024             }
10025         }
10026     }

```

```

10027         mutex_exit(&svntr_hashtab[hash].tr_lock);
10028     }
10029 }
10030 static void
10031 segvn_trupdate_seg(struct seg *seg,
10032     segvn_data_t *svd,
10033     svntr_t *svntrp,
10034     ulong_t hash)
10035 {
10036     proc_t             *p;
10037     lgrp_id_t          lgrp_id;
10038     struct as           *as;
10039     size_t              size;
10040     struct anon_map    *amp;
10041
10042     ASSERT(svd->vp != NULL);
10043     ASSERT(svd->vp == svntrp->tr_vp);
10044     ASSERT(svd->offset == svntrp->tr_off);
10045     ASSERT(svd->offset + seg->s_size == svntrp->tr_eoff);
10046     ASSERT(seg != NULL);
10047     ASSERT(svd->seg == seg);
10048     ASSERT(seg->s_data == (void *)svd);
10049     ASSERT(seg->s_szc == svntrp->tr_szc);
10050     ASSERT(svd->tr_state == SEGVN_TR_ON);
10051     ASSERT(!HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
10052     ASSERT(svd->amp != NULL);
10053     ASSERT(svd->tr_policy_info.mem_policy == LGRP_MEM_POLICY_NEXT_SEG);
10054     ASSERT(svd->tr_policy_info.mem_lgrp_id != LGRP_NONE);
10055     ASSERT(svd->tr_policy_info.mem_lgrp_id < NLGRPS_MAX);
10056     ASSERT(svntrp->tr_amp[svd->tr_policy_info.mem_lgrp_id] == svd->amp);
10057     ASSERT(svntrp->tr_refcnt != 0);
10058     ASSERT(mutex_owned(&svntr_hashtab[hash].tr_lock));
10059
10060     as = seg->s_as;
10061     ASSERT(as != NULL && as != &kas);
10062     p = as->a_proc;
10063     ASSERT(p != NULL);
10064     ASSERT(p->p_tr_lgrp_id != LGRP_NONE);
10065     lgrp_id = p->p_t1_lgrp_id;
10066     if (lgrp_id == LGRP_NONE) {
10067         return;
10068     }
10069     ASSERT(lgrp_id < NLGRPS_MAX);
10070     if (svd->tr_policy_info.mem_lgrp_id == lgrp_id) {
10071         return;
10072     }
10073
10074     /*
10075      * Use tryenter locking since we are locking as/seg and svntr hash
10076      * lock in reverse from syncronous thread order.
10077      */
10078     if (!AS_LOCK_TRYENTER(as, &as->a_lock, RW_READER)) {
10079         SEGVN_TR_ADDSTAT(nolock);
10080         if (segvn_lgrp_trthr_migrs_snpsht) {
10081             segvn_lgrp_trthr_migrs_snpsht = 0;
10082         }
10083         return;
10084     }
10085     if (!SEGVN_LOCK_TRYENTER(seg->s_as, &svd->lock, RW_WRITER)) {
10086         AS_LOCK_EXIT(as, &as->a_lock);
10087         SEGVN_TR_ADDSTAT(nolock);
10088         if (segvn_lgrp_trthr_migrs_snpsht) {
10089             segvn_lgrp_trthr_migrs_snpsht = 0;
10090         }
10091         return;
10092     }

```

```

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 = anommap_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->hat, seg->s_base, size, 0, NULL);
10134     mutex_enter(&svntr_hashtab[hash].tr_lock);
10135
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_lgrp_id != lgrp_id);
10140     ASSERT(svd->amp != svntrp->tr_amp[lgrp_id]);
10141
10142     svd->tr_policy_info.mem_lgrp_id = lgrp_id;
10143     svd->amp = svntrp->tr_amp[lgrp_id];
10144     p->p_tr_lgrp_id = NLGRPS_MAX;
10145     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
10146     AS_LOCK_EXIT(as, &as->a_lock);
10147
10148     ASSERT(svntrp->tr_refcnt != 0);
10149     ASSERT(svd->vp == svntrp->tr_vp);
10150     ASSERT(svd->tr_policy_info.mem_lgrp_id == 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);
10154
10155     SEGVN_TR_ADDSTAT(asyncrepl);
10156 }

```