

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

1

```
*****
280113 Fri May 8 18:05:13 2015
new/usr/src/uts/common/vm/seg_vn.c
PVN_GETPAGE_{SZ,NUM} are misnamed and unnecessarily complicated
There is really no reason to not allow 8 pages all the time. With the
current logic, we get the following:
Assuming 4kB pages (x86):
    _SZ = ptob(8) /* 32kB */
    _NUM = 8
Assuming 8kB pages (sparc):
    _SZ = ptob(8) /* 64kB */
    _NUM = 8
We'd have to deal with 16kB base pages in order for the _NUM #define to not
be 8 (it'd be 4 in that case). So, in the spirit of simplicity, let's just
always grab 8 pages as there are no interesting systems with 16kB+ base pages.
Finally, the defines are poorly named.
*****
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, Joyent, Inc. All rights reserved.
24 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
25 */
27 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
28 /*      All Rights Reserved      */

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 */

40 /*
41 * VM - shared or copy-on-write from a vnode/anonymous memory.
42 */

44 #include <sys/types.h>
45 #include <sys/param.h>
46 #include <sys/t_lock.h>
47 #include <sys/errno.h>
48 #include <sys/systm.h>
49 #include <sys/mman.h>
```

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

2

```
50 #include <sys/debug.h>
51 #include <sys/cred.h>
52 #include <sys/vmsystem.h>
53 #include <sys/tuneable.h>
54 #include <sys/bitmap.h>
55 #include <sys/swap.h>
56 #include <sys/kmem.h>
57 #include <sys/sysmacros.h>
58 #include <sys/vtrace.h>
59 #include <sys/cmn_err.h>
60 #include <sys/callb.h>
61 #include <sys/vm.h>
62 #include <sys/dumphdr.h>
63 #include <sys/lgrp.h>

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>

79 /*
80  * segvn_fault needs a temporary page list array. To avoid calling kmem all
81  * the time, it creates a small (FAULT_TMP_PAGES_NUM entry) array and uses
82  * it if it can. In the rare case when this page list is not large enough,
83  * it goes and gets a large enough array from kmem.
84  * the time, it creates a small (PVN_GETPAGE_NUM entry) array and uses it if
85  * it can. In the rare case when this page list is not large enough, it
86  * goes and gets a large enough array from kmem.
87  *
88  * This small page list array covers either 8 pages or 64kB worth of pages -
89  * whichever is smaller.
90 */
91 #define FAULT_TMP_PAGES_NUM    0x8
92 #define FAULT_TMP_PAGES_SZ    ptob(FAULT_TMP_PAGES_NUM)
93 #define PVN_MAX_GETPAGE_SZ    0x10000
94 #define PVN_MAX_GETPAGE_NUM    0x8

95 #if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
96 #define PVN_GETPAGE_SZ    ptob(PVN_MAX_GETPAGE_NUM)
97 #define PVN_GETPAGE_NUM    PVN_MAX_GETPAGE_NUM
98 #else
99 #define PVN_GETPAGE_SZ    PVN_MAX_GETPAGE_SZ
100 #define PVN_GETPAGE_NUM    btop(PVN_MAX_GETPAGE_SZ)
101 #endif

102 /*
103  * Private seg op routines.
104 */
105 static int    segvn_dup(struct seg *seg, struct seg *newseg);
106 static int    segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
107 static void    segvn_free(struct seg *seg);
108 static faultcode_t    segvn_fault(struct hat *hat, struct seg *seg,
109     caddr_t addr, size_t len, enum fault_type type,
110     enum seg_rw rw);
111 static faultcode_t    segvn_faulta(struct seg *seg, caddr_t addr);
112 static int    segvn_setprot(struct seg *seg, caddr_t addr,
113     size_t len, uint_t prot);
```

```

100 static int      segvn_checkprot(struct seg *seg, caddr_t addr,
101                               size_t len, uint_t prot);
102 static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
103 static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
104                            int attr, uint_t flags);
105 static size_t   segvn_incore(struct seg *seg, caddr_t addr, size_t len,
106                             char *vec);
107 static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
108                              int attr, int op, ulong_t *lockmap, size_t pos);
109 static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
110                               uint_t *protv);
111 static u_offset_t segvn_getoffset(struct seg *seg, caddr_t addr);
112 static int      segvn_gettype(struct seg *seg, caddr_t addr);
113 static int      segvn_getvp(struct seg *seg, caddr_t addr,
114                             struct vnode **vpp);
115 static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,
116                              uint_t behav);
117 static void     segvn_dump(struct seg *seg);
118 static int      segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
119                               struct page **ppp, enum lock_type type, enum seg_rw rw);
120 static int      segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
121                                   uint_t szc);
122 static int      segvn_getmemid(struct seg *seg, caddr_t addr,
123                               memid_t *memidp);
124 static lgrp_mem_policy_info_t *segvn_getpolicy(struct seg *, caddr_t);
125 static int      segvn_inherit(struct seg *, caddr_t, size_t, uint_t);

```

```

127 const struct seg_ops segvn_ops = {
128     .dup          = segvn_dup,
129     .unmap        = segvn_unmap,
130     .free         = segvn_free,
131     .fault        = segvn_fault,
132     .faulta       = segvn_faulta,
133     .setprot      = segvn_setprot,
134     .checkprot    = segvn_checkprot,
135     .kluster      = segvn_kluster,
136     .sync         = segvn_sync,
137     .incore       = segvn_incore,
138     .lockop       = segvn_lockop,
139     .getprot      = segvn_getprot,
140     .getoffset    = segvn_getoffset,
141     .gettype      = segvn_gettype,
142     .getvp        = segvn_getvp,
143     .advise       = segvn_advise,
144     .dump         = segvn_dump,
145     .pagelock     = segvn_pagelock,
146     .setpagesize  = segvn_setpagesize,
147     .getmemid     = segvn_getmemid,
148     .getpolicy    = segvn_getpolicy,
149     .inherit      = segvn_inherit,
150 };

```

unchanged portion omitted

```

4883 int fltadvise = 1;      /* set to free behind pages for sequential access */

4885 /*
4886  * This routine is called via a machine specific fault handling routine.
4887  * It is also called by software routines wishing to lock or unlock
4888  * a range of addresses.
4889  *
4890  * Here is the basic algorithm:
4891  *   If unlocking
4892  *     Call segvn_softunlock
4893  *     Return
4894  *   endif
4895  *   Checking and set up work

```

```

4896 *       If we will need some non-anonymous pages
4897 *       Call VOP_GETPAGE over the range of non-anonymous pages
4898 *       endif
4899 *       Loop over all addresses requested
4900 *       Call segvn_faultpage passing in page list
4901 *       to load up translations and handle anonymous pages
4902 *       endloop
4903 *       Load up translation to any additional pages in page list not
4904 *       already handled that fit into this segment
4905 */
4906 static faultcode_t
4907 segvn_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
4908             enum fault_type type, enum seg_rw rw)
4909 {
4910     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4911     page_t **plp, **ppp, *pp;
4912     u_offset_t off;
4913     caddr_t a;
4914     struct vpage *vpage;
4915     uint_t vpprot, prot;
4916     int err;
4917     page_t *pl[FAULT_TMP_PAGES_NUM + 1];
4918     page_t *pl[PVN_GETPAGE_NUM + 1];
4919     size_t plsz, pl_alloc_sz;
4920     size_t page;
4921     ulong_t anon_index;
4922     struct anon_map *amp;
4923     int dogetpage = 0;
4924     caddr_t lpgaddr, lpgeaddr;
4925     size_t pgsz;
4926     anon_sync_obj_t cookie;
4927     int brkcow = BREAK_COW_SHARE(rw, type, svd->type);

4928     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
4929     ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);

4931     /*
4932     * First handle the easy stuff
4933     */
4934     if (type == F_SOFTUNLOCK) {
4935         if (rw == S_READ_NOCOW) {
4936             rw = S_READ;
4937             ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
4938         }
4939         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4940         pgsz = (seg->s_szc == 0) ? PAGE_SIZE :
4941             page_get_pagesize(seg->s_szc);
4942         VM_STAT_COND_ADD(pgsz > PAGE_SIZE, segvnmstats.fltanpages[16]);
4943         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
4944         segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
4945         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4946         return (0);
4947     }

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

```

```

4961         svd->amp != NULL);
4962     }
4963     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4964 }
4965 } else if (svd->tr_state != SEGVN_TR_OFF) {
4966     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4967
4968     if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
4969         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
4970         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4971         return (FC_PROT);
4972     }
4973
4974     if (svd->tr_state == SEGVN_TR_ON) {
4975         ASSERT(svd->vp != NULL && svd->amp != NULL);
4976         segvn_textunrepl(seg, 0);
4977         ASSERT(svd->amp == NULL &&
4978             svd->tr_state == SEGVN_TR_OFF);
4979     } else if (svd->tr_state != SEGVN_TR_OFF) {
4980         svd->tr_state = SEGVN_TR_OFF;
4981     }
4982     ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
4983     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4984 }
4985
4986 top:
4987 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4988
4989 /*
4990  * If we have the same protections for the entire segment,
4991  * insure that the access being attempted is legitimate.
4992  */
4993
4994 if (svd->pageprot == 0) {
4995     uint_t protchk;
4996
4997     switch (rw) {
4998     case S_READ:
4999     case S_READ_NOCOW:
5000         protchk = PROT_READ;
5001         break;
5002     case S_WRITE:
5003         protchk = PROT_WRITE;
5004         break;
5005     case S_EXEC:
5006         protchk = PROT_EXEC;
5007         break;
5008     case S_OTHER:
5009     default:
5010         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
5011         break;
5012     }
5013
5014     if ((svd->prot & protchk) == 0) {
5015         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5016         return (FC_PROT); /* illegal access type */
5017     }
5018 }
5019
5020 if (brkcow && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5021     /* this must be SOFTLOCK S_READ fault */
5022     ASSERT(svd->amp == NULL);
5023     ASSERT(svd->tr_state == SEGVN_TR_OFF);
5024     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5025     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5026     if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {

```

```

5027     /*
5028     * this must be the first ever non S_READ_NOCOW
5029     * softlock for this segment.
5030     */
5031     ASSERT(svd->softlockcnt == 0);
5032     hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5033         HAT_REGION_TEXT);
5034     svd->rcookie = HAT_INVALID_REGION_COOKIE;
5035 }
5036 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5037 goto top;
5038 }
5039
5040 /*
5041  * We can't allow the long term use of softlocks for vmpss segments,
5042  * because in some file truncation cases we should be able to demote
5043  * the segment, which requires that there are no softlocks. The
5044  * only case where it's ok to allow a SOFTLOCK fault against a vmpss
5045  * segment is S_READ_NOCOW, where the caller holds the address space
5046  * locked as writer and calls softunlock before dropping the as lock.
5047  * S_READ_NOCOW is used by /proc to read memory from another user.
5048  *
5049  * Another deadlock between SOFTLOCK and file truncation can happen
5050  * because segvn_fault_vnodepages() calls the FS one pagesize at
5051  * a time. A second VOP_GETPAGE() call by segvn_fault_vnodepages()
5052  * can cause a deadlock because the first set of page_t's remain
5053  * locked SE_SHARED. To avoid this, we demote segments on a first
5054  * SOFTLOCK if they have a length greater than the segment's
5055  * page size.
5056  *
5057  * So for now, we only avoid demoting a segment on a SOFTLOCK when
5058  * the access type is S_READ_NOCOW and the fault length is less than
5059  * or equal to the segment's page size. While this is quite restrictive,
5060  * it should be the most common case of SOFTLOCK against a vmpss
5061  * segment.
5062  *
5063  * For S_READ_NOCOW, it's safe not to do a copy on write because the
5064  * caller makes sure no COW will be caused by another thread for a
5065  * softlocked page.
5066  */
5067 if (type == F_SOFTLOCK && svd->vp != NULL && seg->s_szc != 0) {
5068     int demote = 0;
5069
5070     if (rw != S_READ_NOCOW) {
5071         demote = 1;
5072     }
5073     if (!demote && len > PAGE_SIZE) {
5074         pgsz = page_get_pagesize(seg->s_szc);
5075         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
5076             lpgeaddr);
5077         if (lpgeaddr - lpgaddr > pgsz) {
5078             demote = 1;
5079         }
5080     }
5081
5082     ASSERT(demote || AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
5083
5084     if (demote) {
5085         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5086         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5087         if (seg->s_szc != 0) {
5088             segvn_vmpss_clrsrc_cnt++;
5089             ASSERT(svd->softlockcnt == 0);
5090             err = segvn_clrsrc(seg);
5091             if (err) {
5092                 segvn_vmpss_clrsrc_err++;

```

```

5093             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5094             return (FC_MAKE_ERR(err));
5095         }
5096     }
5097     ASSERT(seg->s_szc == 0);
5098     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5099     goto top;
5100 }
5101
5102 /*
5103  * Check to see if we need to allocate an anon_map structure.
5104  */
5105 if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
5106     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5107     /*
5108      * Drop the "read" lock on the segment and acquire
5109      * the "write" version since we have to allocate the
5110      * anon_map.
5111      */
5112     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5113     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5114
5115     if (svd->amp == NULL) {
5116         svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
5117         svd->amp->a_szc = seg->s_szc;
5118     }
5119     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5120
5121     /*
5122      * Start all over again since segment protections
5123      * may have changed after we dropped the "read" lock.
5124      */
5125     goto top;
5126 }
5127
5128 /*
5129  * S_READ_NOCOW vs S_READ distinction was
5130  * only needed for the code above. After
5131  * that we treat it as S_READ.
5132  */
5133 if (rw == S_READ_NOCOW) {
5134     ASSERT(type == F_SOFTLOCK);
5135     ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
5136     rw = S_READ;
5137 }
5138
5139 amp = svd->amp;
5140
5141 /*
5142  * MADV_SEQUENTIAL work is ignored for large page segments.
5143  */
5144 if (seg->s_szc != 0) {
5145     pgsz = page_get_pagesize(seg->s_szc);
5146     ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
5147     CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
5148     if (svd->vp == NULL) {
5149         err = segvn_fault_anonpages(hat, seg, lpgaddr,
5150             lpgeaddr, type, rw, addr, addr + len, brkcow);
5151     } else {
5152         err = segvn_fault_vnodepages(hat, seg, lpgaddr,
5153             lpgeaddr, type, rw, addr, addr + len, brkcow);
5154         if (err == IE_RETRY) {
5155             ASSERT(seg->s_szc == 0);
5156             ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
5157             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5158         }
5159     }

```

```

5159         goto top;
5160     }
5161     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5162     return (err);
5163 }
5164
5165 page = seg_page(seg, addr);
5166 if (amp != NULL) {
5167     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5168     anon_index = svd->anon_index + page;
5169
5170     if (type == F_PROT && rw == S_READ &&
5171         svd->tr_state == SEGVN_TR_OFF &&
5172         svd->type == MAP_PRIVATE && svd->pageprot == 0) {
5173         size_t index = anon_index;
5174         struct anon *ap;
5175
5176         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5177         /*
5178          * The fast path could apply to S_WRITE also, except
5179          * that the protection fault could be caused by lazy
5180          * tlb flush when ro->rw. In this case, the pte is
5181          * RW already. But RO in the other cpu's tlb causes
5182          * the fault. Since hat_chgprot won't do anything if
5183          * pte doesn't change, we may end up faulting
5184          * indefinitely until the RO tlb entry gets replaced.
5185          */
5186         for (a = addr; a < addr + len; a += PAGE_SIZE, index++) {
5187             anon_array_enter(amp, index, &cookie);
5188             ap = anon_get_ptr(amp->ahp, index);
5189             anon_array_exit(&cookie);
5190             if ((ap == NULL) || (ap->an_refcnt != 1)) {
5191                 ANON_LOCK_EXIT(&amp->a_rwlock);
5192                 goto slow;
5193             }
5194         }
5195         hat_chgprot(seg->s_as->a_hat, addr, len, svd->prot);
5196         ANON_LOCK_EXIT(&amp->a_rwlock);
5197         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5198         return (0);
5199     }
5200 }
5201 }
5202 slow:
5203
5204 if (svd->vpage == NULL)
5205     vpage = NULL;
5206 else
5207     vpage = &svd->vpage[page];
5208
5209 off = svd->offset + (uintptr_t)(addr - seg->s_base);
5210
5211 /*
5212  * If MADV_SEQUENTIAL has been set for the particular page we
5213  * are faulting on, free behind all pages in the segment and put
5214  * them on the free list.
5215  */
5216
5217 if ((page != 0) && fltadvise && svd->tr_state != SEGVN_TR_ON) {
5218     struct vpage *vpp;
5219     ulong_t fanon_index;
5220     size_t fpage;
5221     u_offset_t pgofff, fpgoff;
5222     struct vnode *fvp;
5223     struct anon *fap = NULL;

```

```

5225     if (svd->advice == MADV_SEQUENTIAL ||
5226         (svd->pageadvice &&
5227          VPP_ADVICE(vp) == MADV_SEQUENTIAL)) {
5228         pgoff = off - PAGESIZE;
5229         fpage = page - 1;
5230         if (vp != NULL)
5231             vpp = &svd->vpage[fpage];
5232         if (amp != NULL)
5233             fanon_index = svd->anon_index + fpage;
5234
5235     while (pgoff > svd->offset) {
5236         if (svd->advice != MADV_SEQUENTIAL &&
5237             (!svd->pageadvice || (vp &&
5238              VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
5239             break;
5240
5241         /*
5242          * If this is an anon page, we must find the
5243          * correct <vp, offset> for it
5244          */
5245         fap = NULL;
5246         if (amp != NULL) {
5247             ANON_LOCK_ENTER(&amp->a_rwlock,
5248                             RW_READER);
5249             anon_array_enter(amp, fanon_index,
5250                             &cookie);
5251             fap = anon_get_ptr(amp->ahp,
5252                               fanon_index);
5253             if (fap != NULL) {
5254                 swap_xlate(fap, &fvp, &fpgoff);
5255             } else {
5256                 fpgoff = pgoff;
5257                 fvp = svd->vp;
5258             }
5259             anon_array_exit(&cookie);
5260             ANON_LOCK_EXIT(&amp->a_rwlock);
5261         } else {
5262             fpgoff = pgoff;
5263             fvp = svd->vp;
5264         }
5265         if (fvp == NULL)
5266             break; /* XXX */
5267
5268         /*
5269          * Skip pages that are free or have an
5270          * "exclusive" lock.
5271          */
5272         pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
5273         if (pp == NULL)
5274             break;
5275
5276         /*
5277          * We don't need the page_struct_lock to test
5278          * as this is only advisory; even if we
5279          * acquire it someone might race in and lock
5280          * the page after we unlock and before the
5281          * PUTPAGE, then VOP_PUTPAGE will do nothing.
5282          */
5283         if (pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
5284             /*
5285              * Hold the vnode before releasing
5286              * the page lock to prevent it from
5287              * being freed and re-used by some
5288              * other thread.
5289              */
5290             VN_HOLD(fvp);
5291             page_unlock(pp);
5292             /*

```

```

5291             * We should build a page list
5292             * to kluster putpages XXX
5293             */
5294             (void) VOP_PUTPAGE(fvp,
5295                               (offset_t)fpgoff, PAGESIZE,
5296                               (B_DONTNEED|B_FREE|B_ASYNC),
5297                               svd->cred, NULL);
5298             VN_RELE(fvp);
5299         } else {
5300             /*
5301              * XXX - Should the loop terminate if
5302              * the page is 'locked'?
5303              */
5304             page_unlock(pp);
5305         }
5306         --vpp;
5307         --fanon_index;
5308         pgoff -= PAGESIZE;
5309     }
5310 }
5311
5312 }
5313
5314 plp = pl;
5315 *plp = NULL;
5316 pl_alloc_sz = 0;
5317
5318 /*
5319 * See if we need to call VOP_GETPAGE for
5320 * *any* of the range being faulted on.
5321 * We can skip all of this work if there
5322 * was no original vnode.
5323 */
5324 if (svd->vp != NULL) {
5325     u_offset_t vp_off;
5326     size_t vp_len;
5327     struct anon *ap;
5328     vnode_t *vp;
5329
5330     vp_off = off;
5331     vp_len = len;
5332
5333     if (amp == NULL)
5334         dogetpage = 1;
5335     else {
5336         /*
5337          * Only acquire reader lock to prevent amp->ahp
5338          * from being changed. It's ok to miss pages,
5339          * hence we don't do anon_array_enter
5340          */
5341         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5342         ap = anon_get_ptr(amp->ahp, anon_index);
5343
5344         if (len <= PAGESIZE)
5345             /* inline non_anon() */
5346             dogetpage = (ap == NULL);
5347         else
5348             dogetpage = non_anon(amp->ahp, anon_index,
5349                                 &vp_off, &vp_len);
5350         ANON_LOCK_EXIT(&amp->a_rwlock);
5351     }
5352
5353     if (dogetpage) {
5354         enum seg_rw arw;
5355         struct as *as = seg->s_as;
5356         if (len > FAULT_TMP_PAGES_SZ) {

```

```

5367     if (len > ptop((sizeof (pl) / sizeof (pl[0])) - 1)) {
5357         /*
5358          * Page list won't fit in local array,
5359          * allocate one of the needed size.
5360          */
5361         pl_alloc_sz =
5362             (btop(len) + 1) * sizeof (page_t *);
5363         plp = kmem_alloc(pl_alloc_sz, KM_SLEEP);
5364         plp[0] = NULL;
5365         plsz = len;
5366     } else if (rw == S_WRITE && svd->type == MAP_PRIVATE ||
5367             svd->tr_state == SEGVN_TR_ON || rw == S_OTHER ||
5368             (((size_t)(addr + PAGE_SIZE) <
5369              (size_t)(seg->s_base + seg->s_size)) &&
5370              hat_probe(as->a_hat, addr + PAGE_SIZE))) {
5371         /*
5372          * Ask VOP_GETPAGE to return the exact number
5373          * of pages if
5374          * (a) this is a COW fault, or
5375          * (b) this is a software fault, or
5376          * (c) next page is already mapped.
5377          */
5378         plsz = len;
5379     } else {
5380         /*
5381          * Ask VOP_GETPAGE to return adjacent pages
5382          * within the segment.
5383          */
5384         plsz = MIN((size_t)FAULT_TMP_PAGES_SZ, (size_t)
5395             plsz = MIN((size_t)PVN_GETPAGE_SZ, (size_t)
5385             ((seg->s_base + seg->s_size) - addr));
5386         ASSERT((addr + plsz) <=
5387             (seg->s_base + seg->s_size));
5388     }
5390     /*
5391     * Need to get some non-anonymous pages.
5392     * We need to make only one call to GETPAGE to do
5393     * this to prevent certain deadlocking conditions
5394     * when we are doing locking. In this case
5395     * non_anon() should have picked up the smallest
5396     * range which includes all the non-anonymous
5397     * pages in the requested range. We have to
5398     * be careful regarding which rw flag to pass in
5399     * because on a private mapping, the underlying
5400     * object is never allowed to be written.
5401     */
5402     if (rw == S_WRITE && svd->type == MAP_PRIVATE) {
5403         arw = S_READ;
5404     } else {
5405         arw = rw;
5406     }
5407     vp = svd->vp;
5408     TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
5409             "segvn_getpage:seg %p addr %p vp %p",
5410             seg, addr, vp);
5411     err = VOP_GETPAGE(vp, (offset_t)vp_off, vp_len,
5412                     &vpprot, plp, plsz, seg, addr + (vp_off - off), arw,
5413                     svd->cred, NULL);
5414     if (err) {
5415         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5416         segvn_pagelist_rele(plp);
5417         if (pl_alloc_sz)
5418             kmem_free(plp, pl_alloc_sz);
5419         return (FC_MAKE_ERR(err));
5420     }

```

```

5421         if (svd->type == MAP_PRIVATE)
5422             vpprot &= ~PROT_WRITE;
5423     }
5424 }
5426     /*
5427     * N.B. at this time the plp array has all the needed non-anon
5428     * pages in addition to (possibly) having some adjacent pages.
5429     */
5431     /*
5432     * Always acquire the anon_array_lock to prevent
5433     * 2 threads from allocating separate anon slots for
5434     * the same "addr".
5435     *
5436     * If this is a copy-on-write fault and we don't already
5437     * have the anon_array_lock, acquire it to prevent the
5438     * fault routine from handling multiple copy-on-write faults
5439     * on the same "addr" in the same address space.
5440     *
5441     * Only one thread should deal with the fault since after
5442     * it is handled, the other threads can acquire a translation
5443     * to the newly created private page. This prevents two or
5444     * more threads from creating different private pages for the
5445     * same fault.
5446     *
5447     * We grab "serialization" lock here if this is a MAP_PRIVATE segment
5448     * to prevent deadlock between this thread and another thread
5449     * which has soft-locked this page and wants to acquire serial_lock.
5450     * ( bug 4026339 )
5451     *
5452     * The fix for bug 4026339 becomes unnecessary when using the
5453     * locking scheme with per amp rlock and a global set of hash
5454     * lock, anon_array_lock. If we steal a vnode page when low
5455     * on memory and upgrad the page lock through page_rename,
5456     * then the page is PAGE_HANDLED, nothing needs to be done
5457     * for this page after returning from segvn_faultpage.
5458     *
5459     * But really, the page lock should be downgraded after
5460     * the stolen page is page_rename'd.
5461     */
5463     if (amp != NULL)
5464         ANON_LOCK_ENTER(&amp->a_rlock, RW_READER);
5466     /*
5467     * Ok, now loop over the address range and handle faults
5468     */
5469     for (a = addr; a < addr + len; a += PAGE_SIZE, off += PAGE_SIZE) {
5470         err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
5471                               type, rw, brkcow);
5472         if (err) {
5473             if (amp != NULL)
5474                 ANON_LOCK_EXIT(&amp->a_rlock);
5475             if (type == F_SOFTLOCK && a > addr) {
5476                 segvn_softunlock(seg, addr, (a - addr),
5477                                 S_OTHER);
5478             }
5479             SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5480             segvn_pagelist_rele(plp);
5481             if (pl_alloc_sz)
5482                 kmem_free(plp, pl_alloc_sz);
5483             return (err);
5484         }
5485         if (vpage) {
5486             vpage++;

```

```

5487     } else if (svd->vpage) {
5488         page = seg_page(seg, addr);
5489         vpage = &svd->vpage[++page];
5490     }
5491 }

5493 /* Didn't get pages from the underlying fs so we're done */
5494 if (!dogetpage)
5495     goto done;

5497 /*
5498  * Now handle any other pages in the list returned.
5499  * If the page can be used, load up the translations now.
5500  * Note that the for loop will only be entered if "plp"
5501  * is pointing to a non-NULL page pointer which means that
5502  * VOP_GETPAGE() was called and vpprot has been initialized.
5503  */
5504 if (svd->pageprot == 0)
5505     prot = svd->prot & vpprot;

5508 /*
5509  * Large Files: diff should be unsigned value because we started
5510  * supporting > 2GB segment sizes from 2.5.1 and when a
5511  * large file of size > 2GB gets mapped to address space
5512  * the diff value can be > 2GB.
5513  */

5515 for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
5516     size_t diff;
5517     struct anon *ap;
5518     int anon_index;
5519     anon_sync_obj_t cookie;
5520     int hat_flag = HAT_LOAD_ADV;

5522     if (svd->flags & MAP_TEXT) {
5523         hat_flag |= HAT_LOAD_TEXT;
5524     }

5526     if (pp == PAGE_HANDLED)
5527         continue;

5529     if (svd->tr_state != SEGVN_TR_ON &&
5530         pp->p_offset >= svd->offset &&
5531         pp->p_offset < svd->offset + seg->s_size) {

5533         diff = pp->p_offset - svd->offset;

5535         /*
5536          * Large Files: Following is the assertion
5537          * validating the above cast.
5538          */
5539         ASSERT(svd->vp == pp->p_vnode);

5541         page = btop(diff);
5542         if (svd->pageprot)
5543             prot = VPP_PROT(&svd->vpage[page]) & vpprot;

5545         /*
5546          * Prevent other threads in the address space from
5547          * creating private pages (i.e., allocating anon slots)
5548          * while we are in the process of loading translations
5549          * to additional pages returned by the underlying
5550          * object.
5551          */
5552         if (amp != NULL) {

```

```

5553         anon_index = svd->anon_index + page;
5554         anon_array_enter(amp, anon_index, &cookie);
5555         ap = anon_get_ptr(amp->ahp, anon_index);
5556     }
5557     if ((amp == NULL) || (ap == NULL)) {
5558         if (IS_VMODSORT(pp->p_vnode) ||
5559             enable_mbit_wa) {
5560             if (rw == S_WRITE)
5561                 hat_setmod(pp);
5562             else if (rw != S_OTHER &&
5563                 !hat_ismod(pp))
5564                 prot &= ~PROT_WRITE;
5565         }
5566         /*
5567          * Skip mapping read ahead pages marked
5568          * for migration, so they will get migrated
5569          * properly on fault
5570          */
5571         ASSERT(amp == NULL ||
5572             svd->rcookie == HAT_INVALID_REGION_COOKIE);
5573         if ((prot & PROT_READ) && !PP_ISMIGRATE(pp)) {
5574             hat_memload_region(hat,
5575                 seg->s_base + diff,
5576                 pp, prot, hat_flag,
5577                 svd->rcookie);
5578         }
5579     }
5580     if (amp != NULL)
5581         anon_array_exit(&cookie);
5582     }
5583     page_unlock(pp);
5584 }
5585 done:
5586     if (amp != NULL)
5587         ANON_LOCK_EXIT(&amp->a_rwlock);
5588     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5589     if (pl_alloc_sz)
5590         kmem_free(plp, pl_alloc_sz);
5591     return (0);
5592 }

```

unchanged portion omitted