

```

*****
286025 Tue Jan 15 10:32:08 2019
new/usr/src/uts/common/vm/seg_vn.c
10095 unchecked return value in segvn_pagelock()
*****
_____unchanged_portion_omitted_____

8761 #ifdef DEBUG
8762 static uint32_t segvn_pglock_mtbf = 0;
8763 #endif

8765 #define PCACHE_SHWLIST      ((page_t *)-2)
8766 #define NOPCACHE_SHWLIST   ((page_t *)-1)

8768 /*
8769  * Lock/Unlock anon pages over a given range. Return shadow list. This routine
8770  * uses global segment pcache to cache shadow lists (i.e. pp arrays) of pages
8771  * to avoid the overhead of per page locking, unlocking for subsequent IOs to
8772  * the same parts of the segment. Currently shadow list creation is only
8773  * supported for pure anon segments. MAP_PRIVATE segment pcache entries are
8774  * tagged with segment pointer, starting virtual address and length. This
8775  * approach for MAP_SHARED segments may add many pcache entries for the same
8776  * set of pages and lead to long hash chains that decrease pcache lookup
8777  * performance. To avoid this issue for shared segments shared anon map and
8778  * starting anon index are used for pcache entry tagging. This allows all
8779  * segments to share pcache entries for the same anon range and reduces pcache
8780  * chain's length as well as memory overhead from duplicate shadow lists and
8781  * pcache entries.
8782  *
8783  * softlockcnt field in segvn_data structure counts the number of F_SOFTLOCK'd
8784  * pages via segvn_fault() and pagelock'd pages via this routine. But pagelock
8785  * part of softlockcnt accounting is done differently for private and shared
8786  * segments. In private segment case softlock is only incremented when a new
8787  * shadow list is created but not when an existing one is found via
8788  * seg_plookup(). pcache entries have reference count incremented/decremented
8789  * by each seg_plookup()/seg_pinactive() operation. Only entries that have 0
8790  * reference count can be purged (and purging is needed before segment can be
8791  * freed). When a private segment pcache entry is purged segvn_reclaim() will
8792  * decrement softlockcnt. Since in private segment case each of its pcache
8793  * entries only belongs to this segment we can expect that when
8794  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8795  * segment purge will succeed and softlockcnt will drop to 0. In shared
8796  * segment case reference count in pcache entry counts active locks from many
8797  * different segments so we can't expect segment purging to succeed even when
8798  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8799  * segment. To be able to determine when there're no pending pagelocks in
8800  * shared segment case we don't rely on purging to make softlockcnt drop to 0
8801  * but instead softlockcnt is incremented and decremented for every
8802  * segvn_pagelock(L_PAGELOCK/L_PAGEUNLOCK) call regardless if a new shadow
8803  * list was created or an existing one was found. When softlockcnt drops to 0
8804  * this segment no longer has any claims for pached shadow lists and the
8805  * segment can be freed even if there're still active pcache entries
8806  * shared by this segment anon map. Shared segment pcache entries belong to
8807  * anon map and are typically removed when anon map is freed after all
8808  * processes destroy the segments that use this anon map.
8809  */
8810 static int
8811 segvn_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***ppp,
8812               enum lock_type type, enum seg_rw rw)
8813 {
8814     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8815     size_t np;
8816     pgcnt_t adjustpages;
8817     pgcnt_t npages;
8818     ulong_t anon_index;
8819     uint_t protchk = (rw == S_READ) ? PROT_READ : PROT_WRITE;

```

```

8820     uint_t error;
8821     struct anon_map *amp;
8822     pgcnt_t anpgcnt;
8823     struct page **pplist, **pl, *pp;
8824     caddr_t a;
8825     size_t page;
8826     caddr_t lpgaddr, lpgeaddr;
8827     anon_sync_obj_t cookie;
8828     int anlock;
8829     struct anon_map *pamp;
8830     caddr_t paddr;
8831     seg_preclaim_cbfunc_t preclaim_callback;
8832     size_t pgsz;
8833     int use_pcache;
8834     size_t wlen;
8835     uint_t pflags = 0;
8836     int sftlck_sbase = 0;
8837     int sftlck_send = 0;

8839 #ifdef DEBUG
8840     if (type == L_PAGELOCK && segvn_pglock_mtbf) {
8841         hrtime_t ts = gethrtime();
8842         if ((ts % segvn_pglock_mtbf) == 0) {
8843             return (ENOTSUP);
8844         }
8845         if ((ts % segvn_pglock_mtbf) == 1) {
8846             return (EFAULT);
8847         }
8848     }
8849 #endif

8851     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
8852            "segvn_pagelock: start seg %p addr %p", seg, addr);

8854     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
8855     ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

8857     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

8859     /*
8860     * for now we only support pagelock to anon memory. We would have to
8861     * check protections for vnode objects and call into the vnode driver.
8862     * That's too much for a fast path. Let the fault entry point handle
8863     * it.
8864     */
8865     if (svd->vp != NULL) {
8866         if (type == L_PAGELOCK) {
8867             error = ENOTSUP;
8868             goto out;
8869         }
8870         panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
8871     }
8872     if ((amp = svd->amp) == NULL) {
8873         if (type == L_PAGELOCK) {
8874             error = EFAULT;
8875             goto out;
8876         }
8877         panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
8878     }
8879     if (rw != S_READ && rw != S_WRITE) {
8880         if (type == L_PAGELOCK) {
8881             error = ENOTSUP;
8882             goto out;
8883         }
8884         panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
8885     }

```

```

8887     if (seg->s_szc != 0) {
8888         /*
8889          * We are adjusting the pagelock region to the large page size
8890          * boundary because the unlocked part of a large page cannot
8891          * be freed anyway unless all constituent pages of a large
8892          * page are locked. Bigger regions reduce pcache chain length
8893          * and improve lookup performance. The tradeoff is that the
8894          * very first segvn_pagelock() call for a given page is more
8895          * expensive if only 1 page_t is needed for IO. This is only
8896          * an issue if pcache entry doesn't get reused by several
8897          * subsequent calls. We optimize here for the case when pcache
8898          * is heavily used by repeated IOs to the same address range.
8899          *
8900          * Note segment's page size cannot change while we are holding
8901          * as lock. And then it cannot change while softlockcnt is
8902          * not 0. This will allow us to correctly recalculate large
8903          * page size region for the matching pageunlock/reclaim call
8904          * since as_pageunlock() caller must always match
8905          * as_pagelock() call's addr and len.
8906          *
8907          * For pageunlock *ppp points to the pointer of page_t that
8908          * corresponds to the real unadjusted start address. Similar
8909          * for pagelock *ppp must point to the pointer of page_t that
8910          * corresponds to the real unadjusted start address.
8911          */
8912         pgsz = page_get_pagesize(seg->s_szc);
8913         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
8914         adjustpages = btop((uintptr_t)(addr - lpgaddr));
8915     } else if (len < segvn_pglock_comb_thrshld) {
8916         lpgaddr = addr;
8917         lpgeaddr = addr + len;
8918         adjustpages = 0;
8919         pgsz = PAGE_SIZE;
8920     } else {
8921         /*
8922          * Align the address range of large enough requests to allow
8923          * combining of different shadow lists into 1 to reduce memory
8924          * overhead from potentially overlapping large shadow lists
8925          * (worst case is we have a LMB IO into buffers with start
8926          * addresses separated by 4K). Alignment is only possible if
8927          * padded chunks have sufficient access permissions. Note
8928          * permissions won't change between L_PAGELOCK and
8929          * L_PAGEUNLOCK calls since non 0 softlockcnt will force
8930          * segvn_setprot() to wait until softlockcnt drops to 0. This
8931          * allows us to determine in L_PAGEUNLOCK the same range we
8932          * computed in L_PAGELOCK.
8933          *
8934          * If alignment is limited by segment ends set
8935          * sftlck_sbase/sftlck_send flags. In L_PAGELOCK case when
8936          * these flags are set bump softlockcnt_sbase/softlockcnt_send
8937          * per segment counters. In L_PAGEUNLOCK case decrease
8938          * softlockcnt_sbase/softlockcnt_send counters if
8939          * sftlck_sbase/sftlck_send flags are set. When
8940          * softlockcnt_sbase/softlockcnt_send are non 0
8941          * segvn_concat()/segvn_extend_prev()/segvn_extend_next()
8942          * won't merge the segments. This restriction combined with
8943          * restriction on segment unmapping and splitting for segments
8944          * that have non 0 softlockcnt allows L_PAGEUNLOCK to
8945          * correctly determine the same range that was previously
8946          * locked by matching L_PAGELOCK.
8947          */
8948         pflags = SEGP_PSHIFT | (segvn_pglock_comb_bshift << 16);
8949         pgsz = PAGE_SIZE;
8950         if (svd->type == MAP_PRIVATE) {
8951             lpgaddr = (caddr_t)P2ALIGN((uintptr_t)addr,

```

```

8952             segvn_pglock_comb_balign);
8953             if (lpgaddr < seg->s_base) {
8954                 lpgaddr = seg->s_base;
8955                 sftlck_sbase = 1;
8956             }
8957         } else {
8958             ulong_t aix = svd->anon_index + seg_page(seg, addr);
8959             ulong_t aaix = P2ALIGN(aix, segvn_pglock_comb_balign);
8960             if (aaix < svd->anon_index) {
8961                 lpgaddr = seg->s_base;
8962                 sftlck_sbase = 1;
8963             } else {
8964                 lpgaddr = addr - ptob(aix - aaix);
8965                 ASSERT(lpgaddr >= seg->s_base);
8966             }
8967         }
8968     }
8969     if (svd->pageprot && lpgaddr != addr) {
8970         struct vpage *vp = &svd->vpage[seg_page(seg, lpgaddr)];
8971         struct vpage *evp = &svd->vpage[seg_page(seg, addr)];
8972         while (vp < evp) {
8973             if ((VPP_PROT(vp) & protchk) == 0) {
8974                 break;
8975             }
8976             vp++;
8977         }
8978         if (vp < evp) {
8979             lpgaddr = addr;
8980             pflags = 0;
8981         }
8982     }
8983     lpgeaddr = addr + len;
8984     if (pflags) {
8985         if (svd->type == MAP_PRIVATE) {
8986             lpgeaddr = (caddr_t)P2ROUNDUP(
8987                 (uintptr_t)lpgeaddr,
8988                 segvn_pglock_comb_balign);
8989         } else {
8990             ulong_t aix = svd->anon_index +
8991                 seg_page(seg, lpgeaddr);
8992             ulong_t aaix = P2ROUNDUP(aix,
8993                 segvn_pglock_comb_balign);
8994             if (aaix < aix) {
8995                 lpgeaddr = 0;
8996             } else {
8997                 lpgeaddr += ptob(aaix - aix);
8998             }
8999         }
9000         if (lpgeaddr == 0 ||
9001             lpgeaddr > seg->s_base + seg->s_size) {
9002             lpgeaddr = seg->s_base + seg->s_size;
9003             sftlck_send = 1;
9004         }
9005     }
9006     if (svd->pageprot && lpgeaddr != addr + len) {
9007         struct vpage *vp;
9008         struct vpage *evp;
9009
9010         vp = &svd->vpage[seg_page(seg, addr + len)];
9011         evp = &svd->vpage[seg_page(seg, lpgeaddr)];
9012
9013         while (vp < evp) {
9014             if ((VPP_PROT(vp) & protchk) == 0) {
9015                 break;
9016             }
9017             vp++;
9018         }
9019     }

```

```

9018         if (vp < evp) {
9019             lpgeaddr = addr + len;
9020         }
9021     }
9022     adjustpages = btop((uintptr_t)(addr - lpgaddr));
9023 }
9024
9025 /*
9026  * For MAP_SHARED segments we create pcache entries tagged by amp and
9027  * anon index so that we can share pcache entries with other segments
9028  * that map this amp. For private segments pcache entries are tagged
9029  * with segment and virtual address.
9030  */
9031 if (svd->type == MAP_SHARED) {
9032     pamp = amp;
9033     paddr = (caddr_t)((lpgaddr - seg->s_base) +
9034         ptob(svd->anon_index));
9035     preclaim_callback = shamp_reclaim;
9036 } else {
9037     pamp = NULL;
9038     paddr = lpgaddr;
9039     preclaim_callback = segvn_reclaim;
9040 }
9041
9042 if (type == L_PAGEUNLOCK) {
9043     VM_STAT_ADD(segvmstats.pagelock[0]);
9044
9045     /*
9046      * update hat ref bits for /proc. We need to make sure
9047      * that threads tracing the ref and mod bits of the
9048      * address space get the right data.
9049      * Note: page ref and mod bits are updated at reclaim time
9050      */
9051     if (seg->s_as->a_vbits) {
9052         for (a = addr; a < addr + len; a += PAGE_SIZE) {
9053             if (rw == S_WRITE) {
9054                 hat_setstat(seg->s_as, a,
9055                     PAGE_SIZE, P_REF | P_MOD);
9056             } else {
9057                 hat_setstat(seg->s_as, a,
9058                     PAGE_SIZE, P_REF);
9059             }
9060         }
9061     }
9062
9063     /*
9064      * Check the shadow list entry after the last page used in
9065      * this IO request. If it's NOPCACHE_SHWLIST the shadow list
9066      * was not inserted into pcache and is not large page
9067      * adjusted. In this case call reclaim callback directly and
9068      * don't adjust the shadow list start and size for large
9069      * pages.
9070      */
9071     npages = btop(len);
9072     if ((*ppp)[npages] == NOPCACHE_SHWLIST) {
9073         void *ptag;
9074         if (pamp != NULL) {
9075             ASSERT(svd->type == MAP_SHARED);
9076             ptag = (void *)pamp;
9077             paddr = (caddr_t)((addr - seg->s_base) +
9078                 ptob(svd->anon_index));
9079         } else {
9080             ptag = (void *)seg;
9081             paddr = addr;
9082         }
9083         (void) preclaim_callback(ptag, paddr, len, *ppp, rw, 0);

```

```

9083         (*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
9084     } else {
9085         ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
9086             IS_SWAPFSVP((*ppp)[npages]->p_vnode));
9087         len = lpgeaddr - lpgaddr;
9088         npages = btop(len);
9089         seg_pinactive(seg, pamp, paddr, len,
9090             *ppp - adjustpages, rw, pflags, preclaim_callback);
9091     }
9092
9093     if (pamp != NULL) {
9094         ASSERT(svd->type == MAP_SHARED);
9095         ASSERT(svd->softlockcnt >= npages);
9096         atomic_add_long((ulong_t *)&svd->softlockcnt, -npages);
9097     }
9098
9099     if (sftlck_sbase) {
9100         ASSERT(svd->softlockcnt_sbase > 0);
9101         atomic_dec_ulong((ulong_t *)&svd->softlockcnt_sbase);
9102     }
9103     if (sftlck_send) {
9104         ASSERT(svd->softlockcnt_send > 0);
9105         atomic_dec_ulong((ulong_t *)&svd->softlockcnt_send);
9106     }
9107
9108     /*
9109      * If someone is blocked while unmapping, we purge
9110      * segment page cache and thus reclaim plist synchronously
9111      * without waiting for seg_pasync_thread. This speeds up
9112      * unmapping in cases where munmap(2) is called, while
9113      * raw async i/o is still in progress or where a thread
9114      * exits on data fault in a multithreaded application.
9115      */
9116     if (AS_ISUNMAPWAIT(seg->s_as)) {
9117         if (svd->softlockcnt == 0) {
9118             mutex_enter(&seg->s_as->a_contents);
9119             if (AS_ISUNMAPWAIT(seg->s_as)) {
9120                 AS_CLRUNMAPWAIT(seg->s_as);
9121                 cv_broadcast(&seg->s_as->a_cv);
9122             }
9123             mutex_exit(&seg->s_as->a_contents);
9124         } else if (pamp == NULL) {
9125             /*
9126              * softlockcnt is not 0 and this is a
9127              * MAP_PRIVATE segment. Try to purge its
9128              * pcache entries to reduce softlockcnt.
9129              * If it drops to 0 segvn_reclaim()
9130              * will wake up a thread waiting on
9131              * unmapwait flag.
9132              *
9133              * We don't purge MAP_SHARED segments with non
9134              * 0 softlockcnt since IO is still in progress
9135              * for such segments.
9136              */
9137             ASSERT(svd->type == MAP_PRIVATE);
9138             segvn_purge(seg);
9139         }
9140     }
9141     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9142     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_UNLOCK_END,
9143         "segvn_pagelock: unlock seg %p addr %p", seg, addr);
9144     return (0);
9145 }
9146
9147 /* The L_PAGELOCK case ... */

```

```

9149     VM_STAT_ADD(segvmvstats.pagelock[1]);
9151     /*
9152     * For MAP_SHARED segments we have to check protections before
9153     * seg_plookup() since pcache entries may be shared by many segments
9154     * with potentially different page protections.
9155     */
9156     if (pamp != NULL) {
9157         ASSERT(svd->type == MAP_SHARED);
9158         if (svd->pageprot == 0) {
9159             if ((svd->prot & protchk) == 0) {
9160                 error = EACCES;
9161                 goto out;
9162             }
9163         } else {
9164             /*
9165             * check page protections
9166             */
9167             caddr_t ea;

9169             if (seg->s_szc) {
9170                 a = lpgaddr;
9171                 ea = lpgeaddr;
9172             } else {
9173                 a = addr;
9174                 ea = addr + len;
9175             }
9176             for (; a < ea; a += pgsz) {
9177                 struct vpage *vp;

9179                 ASSERT(seg->s_szc == 0 ||
9180                        sameprot(seg, a, pgsz));
9181                 vp = &svd->vpage[seg_page(seg, a)];
9182                 if ((VPP_PROT(vp) & protchk) == 0) {
9183                     error = EACCES;
9184                     goto out;
9185                 }
9186             }
9187         }
9188     }

9190     /*
9191     * try to find pages in segment page cache
9192     */
9193     pplist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
9194     if (pplist != NULL) {
9195         if (pamp != NULL) {
9196             npages = btop((uintptr_t)(lpgeaddr - lpgaddr));
9197             ASSERT(svd->type == MAP_SHARED);
9198             atomic_add_long((ulong_t *)&svd->softlockcnt,
9199                            npages);
9200         }
9201         if (sftlck_sbase) {
9202             atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9203         }
9204         if (sftlck_send) {
9205             atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9206         }
9207         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9208         *ppp = pplist + adjustpages;
9209         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_HIT_END,
9210               "segvn_pagelock: cache hit seg %p addr %p", seg, addr);
9211         return (0);
9212     }

9214     /*

```

```

9215     * For MAP_SHARED segments we already verified above that segment
9216     * protections allow this pagelock operation.
9217     */
9218     if (pamp == NULL) {
9219         ASSERT(svd->type == MAP_PRIVATE);
9220         if (svd->pageprot == 0) {
9221             if ((svd->prot & protchk) == 0) {
9222                 error = EACCES;
9223                 goto out;
9224             }
9225             if (svd->prot & PROT_WRITE) {
9226                 wlen = lpgeaddr - lpgaddr;
9227             } else {
9228                 wlen = 0;
9229                 ASSERT(rw == S_READ);
9230             }
9231         } else {
9232             int wcont = 1;
9233             /*
9234             * check page protections
9235             */
9236             for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
9237                 struct vpage *vp;

9239                 ASSERT(seg->s_szc == 0 ||
9240                        sameprot(seg, a, pgsz));
9241                 vp = &svd->vpage[seg_page(seg, a)];
9242                 if ((VPP_PROT(vp) & protchk) == 0) {
9243                     error = EACCES;
9244                     goto out;
9245                 }
9246                 if (wcont && (VPP_PROT(vp) & PROT_WRITE)) {
9247                     wlen += pgsz;
9248                 } else {
9249                     wcont = 0;
9250                     ASSERT(rw == S_READ);
9251                 }
9252             }
9253         }
9254         ASSERT(rw == S_READ || wlen == lpgeaddr - lpgaddr);
9255         ASSERT(rw == S_WRITE || wlen <= lpgeaddr - lpgaddr);
9256     }

9258     /*
9259     * Only build large page adjusted shadow list if we expect to insert
9260     * it into pcache. For large enough pages it's a big overhead to
9261     * create a shadow list of the entire large page. But this overhead
9262     * should be amortized over repeated pcache hits on subsequent reuse
9263     * of this shadow list (IO into any range within this shadow list will
9264     * find it in pcache since we large page align the request for pcache
9265     * lookups). pcache performance is improved with bigger shadow lists
9266     * as it reduces the time to pcache the entire big segment and reduces
9267     * pcache chain length.
9268     */
9269     if (seg_pinsert_check(seg, pamp, paddr,
9270                          lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
9271         addr = lpgaddr;
9272         len = lpgeaddr - lpgaddr;
9273         use_pcache = 1;
9274     } else {
9275         use_pcache = 0;
9276         /*
9277         * Since this entry will not be inserted into the pcache, we
9278         * will not do any adjustments to the starting address or
9279         * size of the memory to be locked.
9280         */

```

```

9281         adjustpages = 0;
9282     }
9283     npages = btop(len);

9285     pplist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
9286     pl = pplist;
9287     *ppp = pplist + adjustpages;
9288     /*
9289     * If use_pcache is 0 this shadow list is not large page adjusted.
9290     * Record this info in the last entry of shadow array so that
9291     * L_PAGEUNLOCK can determine if it should large page adjust the
9292     * address range to find the real range that was locked.
9293     */
9294     pl[npages] = use_pcache ? PCACHE_SHWLIST : NOPCACHE_SHWLIST;

9296     page = seg_page(seg, addr);
9297     anon_index = svd->anon_index + page;

9299     anlock = 0;
9300     ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
9301     ASSERT(amp->a_szc >= seg->s_szc);
9302     anpgcnt = page_get_pagecnt(amp->a_szc);
9303     for (a = addr; a < addr + len; a += PAGESIZE, anon_index++) {
9304         struct anon *ap;
9305         struct vnode *vp;
9306         u_offset_t off;

9308         /*
9309         * Lock and unlock anon array only once per large page.
9310         * anon_array_enter() locks the root anon slot according to
9311         * a_szc which can't change while anon map is locked. We lock
9312         * anon the first time through this loop and each time we
9313         * reach anon index that corresponds to a root of a large
9314         * page.
9315         */
9316         if (a == addr || P2PHASE(anon_index, anpgcnt) == 0) {
9317             ASSERT(anlock == 0);
9318             anon_array_enter(amp, anon_index, &cookie);
9319             anlock = 1;
9320         }
9321         ap = anon_get_ptr(amp->ahp, anon_index);

9323         /*
9324         * We must never use seg_pcache for COW pages
9325         * because we might end up with original page still
9326         * lying in seg_pcache even after private page is
9327         * created. This leads to data corruption as
9328         * aio_write refers to the page still in cache
9329         * while all other accesses refer to the private
9330         * page.
9331         */
9332         if (ap == NULL || ap->an_refcnt != 1) {
9333             struct vpage *vpage;

9335             if (seg->s_szc) {
9336                 error = EFAULT;
9337                 break;
9338             }
9339             if (svd->vpage != NULL) {
9340                 vpage = &svd->vpage[seg_page(seg, a)];
9341             } else {
9342                 vpage = NULL;
9343             }
9344             ASSERT(anlock);
9345             anon_array_exit(&cookie);
9346             anlock = 0;

```

```

9347         pp = NULL;
9348         error = segvn_faultpage(seg->s_as->a_hat, seg, a, 0,
9349             vpage, &pp, 0, F_INVALID, rw, 1);
9350         if (error) {
9351             error = fc_decode(error);
9352             break;
9353         }
9354         anon_array_enter(amp, anon_index, &cookie);
9355         anlock = 1;
9356         ap = anon_get_ptr(amp->ahp, anon_index);
9357         if (ap == NULL || ap->an_refcnt != 1) {
9358             error = EFAULT;
9359             break;
9360         }
9361     }
9362     swap_xlate(ap, &vp, &off);
9363     pp = page_lookup_nowait(vp, off, SE_SHARED);
9364     if (pp == NULL) {
9365         error = EFAULT;
9366         break;
9367     }
9368     if (ap->an_pvp != NULL) {
9369         anon_swap_free(ap, pp);
9370     }
9371     /*
9372     * Unlock anon if this is the last slot in a large page.
9373     */
9374     if (P2PHASE(anon_index, anpgcnt) == anpgcnt - 1) {
9375         ASSERT(anlock);
9376         anon_array_exit(&cookie);
9377         anlock = 0;
9378     }
9379     *pplist++ = pp;
9380 }
9381 if (anlock) { /* Ensure the lock is dropped */
9382     anon_array_exit(&cookie);
9383 }
9384 ANON_LOCK_EXIT(&amp->a_rwlock);

9386 if (a >= addr + len) {
9387     atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
9388     if (pamp != NULL) {
9389         ASSERT(svd->type == MAP_SHARED);
9390         atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
9391             npages);
9392         wlen = len;
9393     }
9394     if (sftlck_sbase) {
9395         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9396     }
9397     if (sftlck_send) {
9398         atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9399     }
9400     if (use_pcache) {
9401         (void) seg_pininsert(seg, pamp, paddr, len, wlen, pl,
9402             rw, pflags, preclaim_callback);
9403     }
9404     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9405     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
9406         "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
9407     return (0);
9408 }

9410 pplist = pl;
9411 np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
9412 while (np > (uint_t)0) {

```

```
9413         ASSERT(PAGE_LOCKED(*pplist));
9414         page_unlock(*pplist);
9415         np--;
9416         pplist++;
9417     }
9418     kmem_free(pl, sizeof (page_t *) * (npages + 1));
9419 out:
9420     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9421     *ppp = NULL;
9422     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
9423           "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
9424     return (error);
9425 }
unchanged_portion_omitted
```