****
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_

```
#ifdef DEBUG
static uint32_t segvn_pglock_mtbf = 0;
#endif

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

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



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

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

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

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

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

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

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



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

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

			while (vp < evp) {
				if ((VPP_PROT(vp) & protchk) == 0) {
					break;
				}
				vp++;
			}
```

```
				if (vp < evp) {
					lpgeaddr = addr + len;
				}
			}
			adjustpages = btop((uintptr_t)(addr - lpgaddr));
		}

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

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

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

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



```
				(*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
			} else {
				ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
				    IS_SWAPFSVP((*ppp)[npages]->p_vnode));
				len = lpgeaddr - lpgaddr;
				npages = btop(len);
				seg_pinactive(seg, pamp, paddr, len,
				    *ppp - adjustpages, rw, pflags, preclaim_callback);
			}

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

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

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

		/* The L_PAGELOCK case ... */
```

```c
        VM_STAT_ADD(segvnvmstats.pagelock[1]);

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

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

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

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

        /*
```



```c
         * For MAP_SHARED segments we already verified above that segment
         * protections allow this pagelock operation.
         */
        if (pamp == NULL) {
                ASSERT(svd->type == MAP_PRIVATE);
                if (svd->pageprot == 0) {
                        if ((svd->prot & protchk) == 0) {
                                error = EACCES;
                                goto out;
                        }
                        if (svd->prot & PROT_WRITE) {
                                wlen = lpgeaddr - lpgaddr;
                        } else {
                                wlen = 0;
                                ASSERT(rw == S_READ);
                        }
                } else {
                        int wcont = 1;
                        /*
                         * check page protections
                         */
                        for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
                                struct vpage *vp;

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

        /*
         * Only build large page adjusted shadow list if we expect to insert
         * it into pcache. For large enough pages it’s a big overhead to
         * create a shadow list of the entire large page. But this overhead
         * should be amortized over repeated pcache hits on subsequent reuse
         * of this shadow list (IO into any range within this shadow list will
         * find it in pcache since we large page align the request for pcache
         * lookups). pcache performance is improved with bigger shadow lists
         * as it reduces the time to pcache the entire big segment and reduces
         * pcache chain length.
         */
        if (seg_pinsert_check(seg, pamp, paddr,
            lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
                addr = lpgaddr;
                len = lpgeaddr - lpgaddr;
                use_pcache = 1;
        } else {
                use_pcache = 0;
                /*
                 * Since this entry will not be inserted into the pcache, we
                 * will not do any adjustments to the starting address or
                 * size of the memory to be locked.
                 */
```

```
                adjustpages = 0;
        }
        npages = btop(len);

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

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

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

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

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

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



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

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

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

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

______***unchanged_portion_omitted_***