```
**********************************************************
   10256 Tue Nov 24 09:34:41 2015
new/usr/src/uts/common/vm/seg.h
6145 instead using SEGOP_* macros, define full-fledged segop_* functions
**********************************************************
______unchanged_portion_omitted_
```

```
#ifdef _KERNEL

/*
 * Generic segment operations
 */
extern  void    seg_init(void);
extern  struct  seg *seg_alloc(struct as *as, caddr_t base, size_t size);
extern  int     seg_attach(struct as *as, caddr_t base, size_t size,
                        struct seg *seg);
extern  void    seg_unmap(struct seg *seg);
extern  void    seg_free(struct seg *seg);

/*
 * functions for pagelock cache support
 */
typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
    struct page **, enum seg_rw, int);

extern  struct  page **seg_plookup(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);
extern  void    seg_pinactive(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,
    uint_t flags, seg_preclaim_cbfunc_t callback);

extern  void    seg_ppurge(struct seg *seg, struct anon_map *amp,
    uint_t flags);
extern  void    seg_ppurge_wiredpp(struct page **pp);

extern  int     seg_pinsert_check(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, uint_t flags);
extern  int     seg_pinsert(struct seg *seg, struct anon_map *amp,
    caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,
    uint_t flags, seg_preclaim_cbfunc_t callback);

extern  void    seg_pasync_thread(void);
extern  void    seg_preap(void);
extern  int     seg_p_disable(void);
extern  void    seg_p_enable(void);

extern  segadvstat_t    segadvstat;

/*
 * Flags for pagelock cache support.
 * Flags argument is passed as uint_t to pcache routines.  upper 16 bits of
 * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
 * is set.
 */
#define SEGP_FORCE_WIRED        0x1     /* skip check against seg_pwindow */
#define SEGP_AMP                0x2     /* anon map's pcache entry */
#define SEGP_PSHIFT             0x4     /* addr pgsz shift for hash function */

/*
 * Return values for seg_pinsert and seg_pinsert_check functions.
 */
#define SEGP_SUCCESS            0       /* seg_pinsert() succeeded */
#define SEGP_FAIL               1       /* seg_pinsert() failed */

/* Page status bits for segop_incore */
#define SEG_PAGE_INCORE         0x01    /* VA has a page backing it */
```

```
#define SEG_PAGE_LOCKED         0x02    /* VA has a page that is locked */
#define SEG_PAGE_HASCOW         0x04    /* VA has a page with a copy-on-write */
#define SEG_PAGE_SOFTLOCK       0x08    /* VA has a page with softlock held */
#define SEG_PAGE_VNODEBACKED    0x10    /* Segment is backed by a vnode */
#define SEG_PAGE_ANON           0x20    /* VA has an anonymous page */
#define SEG_PAGE_VNODE          0x40    /* VA has a vnode page backing it */

#define SEGOP_DUP(s, n)             (*(s)->s_ops->dup)((s), (n))
#define SEGOP_UNMAP(s, a, l)        (*(s)->s_ops->unmap)((s), (a), (l))
#define SEGOP_FREE(s)               (*(s)->s_ops->free)((s))
#define SEGOP_FAULT(h, s, a, l, t, rw) \
                (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
#define SEGOP_FAULTA(s, a)          (*(s)->s_ops->faulta)((s), (a))
#define SEGOP_SETPROT(s, a, l, p)   (*(s)->s_ops->setprot)((s), (a), (l), (p))
#define SEGOP_CHECKPROT(s, a, l, p) (*(s)->s_ops->checkprot)((s), (a), (l), (p))
#define SEGOP_KLUSTER(s, a, d)      (*(s)->s_ops->kluster)((s), (a), (d))
#define SEGOP_SWAPOUT(s)            (*(s)->s_ops->swapout)((s))
#define SEGOP_SYNC(s, a, l, atr, f) \
                (*(s)->s_ops->sync)((s), (a), (l), (atr), (f))
#define SEGOP_INCORE(s, a, l, v)    (*(s)->s_ops->incore)((s), (a), (l), (v))
#define SEGOP_LOCKOP(s, a, l, atr, op, b, p) \
                (*(s)->s_ops->lockop)((s), (a), (l), (atr), (op), (b), (p))
#define SEGOP_GETPROT(s, a, l, p)   (*(s)->s_ops->getprot)((s), (a), (l), (p))
#define SEGOP_GETOFFSET(s, a)       (*(s)->s_ops->getoffset)((s), (a))
#define SEGOP_GETTYPE(s, a)         (*(s)->s_ops->gettype)((s), (a))
#define SEGOP_GETVP(s, a, vpp)      (*(s)->s_ops->getvp)((s), (a), (vpp))
#define SEGOP_ADVISE(s, a, l, b)    (*(s)->s_ops->advise)((s), (a), (l), (b))
#define SEGOP_DUMP(s)               (*(s)->s_ops->dump)((s))
#define SEGOP_PAGELOCK(s, a, l, p, t, rw) \
                (*(s)->s_ops->pagelock)((s), (a), (l), (p), (t), (rw))
#define SEGOP_SETPAGESIZE(s, a, l, szc) \
                (*(s)->s_ops->setpagesize)((s), (a), (l), (szc))
#define SEGOP_GETMEMID(s, a, mp)    (*(s)->s_ops->getmemid)((s), (a), (mp))
#define SEGOP_GETPOLICY(s, a)       (*(s)->s_ops->getpolicy)((s), (a))
#define SEGOP_CAPABLE(s, c)         (*(s)->s_ops->capable)((s), (c))
#define SEGOP_INHERIT(s, a, l, b)   (*(s)->s_ops->inherit)((s), (a), (l), (b))

#define seg_page(seg, addr) \
        (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)

#define seg_pages(seg) \
        (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)

#define IE_NOMEM        -1      /* internal to seg layer */
#define IE_RETRY        -2      /* internal to seg layer */
#define IE_REATTACH     -3      /* internal to seg layer */

/* Values for SEGOP_INHERIT */
#define SEGP_INH_ZERO   0x01

int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);

/* Delay/retry factors for seg_p_mem_config_pre_del */
#define SEGP_PREDEL_DELAY_FACTOR        4
/*
 * As a workaround to being unable to purge the pagelock
 * cache during a DR delete memory operation, we use
 * a stall threshold that is twice the maximum seen
 * during testing.  This workaround will be removed
 * when a suitable fix is found.
 */
#define SEGP_STALL_SECONDS      25
#define SEGP_STALL_THRESHOLD \
        (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

#ifdef VMDEBUG
```

```
uint_t  seg_page(struct seg *, caddr_t);
uint_t  seg_pages(struct seg *);

#endif  /* VMDEBUG */

boolean_t       seg_can_change_zones(struct seg *);
size_t          seg_swresv(struct seg *);

/* segop wrappers */
extern int segop_dup(struct seg *, struct seg *);
extern int segop_unmap(struct seg *, caddr_t, size_t);
extern void segop_free(struct seg *);
extern faultcode_t segop_fault(struct hat *, struct seg *, caddr_t, size_t,
    enum fault_type, enum seg_rw);
extern faultcode_t segop_faulta(struct seg *, caddr_t);
extern int segop_setprot(struct seg *, caddr_t, size_t, uint_t);
extern int segop_checkprot(struct seg *, caddr_t, size_t, uint_t);
extern int segop_kluster(struct seg *, caddr_t, ssize_t);
extern size_t segop_swapout(struct seg *);
extern int segop_sync(struct seg *, caddr_t, size_t, int, uint_t);
extern size_t segop_incore(struct seg *, caddr_t, size_t, char *);
extern int segop_lockop(struct seg *, caddr_t, size_t, int, int, ulong_t *,
    size_t);
extern int segop_getprot(struct seg *, caddr_t, size_t, uint_t *);
extern u_offset_t segop_getoffset(struct seg *, caddr_t);
extern int segop_gettype(struct seg *, caddr_t);
extern int segop_getvp(struct seg *, caddr_t, struct vnode **);
extern int segop_advise(struct seg *, caddr_t, size_t, uint_t);
extern void segop_dump(struct seg *);
extern int segop_pagelock(struct seg *, caddr_t, size_t, struct page ***,
    enum lock_type, enum seg_rw);
extern int segop_setpagesize(struct seg *, caddr_t, size_t, uint_t);
extern int segop_getmemid(struct seg *, caddr_t, memid_t *);
extern struct lgrp_mem_policy_info *segop_getpolicy(struct seg *, caddr_t);
extern int segop_capable(struct seg *, segcapability_t);
extern int segop_inherit(struct seg *, caddr_t, size_t, uint_t);
#endif /* ! codereview */

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
#endif

#endif  /* _VM_SEG_H */
```

```
**********************************************************
   54541 Tue Nov 24 09:34:42 2015
new/usr/src/uts/common/vm/vm_seg.c
6145 instead using SEGOP_* macros, define full-fledged segop_* functions
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright (c) 2015, Joyent, Inc.
 * Copyright 2015, Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - segment management.
 */

#include <sys/types.h>
#include <sys/inttypes.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/debug.h>
#include <sys/fs/swapnode.h>
#include <sys/cmn_err.h>
#include <sys/callb.h>
#include <sys/mem_config.h>
#include <sys/mman.h>
```

```
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <vm/seg_spt.h>
#include <vm/seg_vn.h>
#include <vm/anon.h>

/*
 * kstats for segment advise
 */
segadvstat_t segadvstat = {
        { "MADV_FREE_hit",      KSTAT_DATA_ULONG },
        { "MADV_FREE_miss",     KSTAT_DATA_ULONG },
};

kstat_named_t *segadvstat_ptr = (kstat_named_t *)&segadvstat;
uint_t segadvstat_ndata = sizeof (segadvstat) / sizeof (kstat_named_t);

/*
 * entry in the segment page cache
 */
struct seg_pcache {
        struct seg_pcache       *p_hnext;       /* list for hashed blocks */
        struct seg_pcache       *p_hprev;
        pcache_link_t           p_plink;        /* per segment/amp list */
        void                    *p_htag0;       /* segment/amp pointer */
        caddr_t                 p_addr;         /* base address/anon_idx */
        size_t                  p_len;          /* total bytes */
        size_t                  p_wlen;         /* writtable bytes at p_addr */
        struct page             **p_pp;         /* pp shadow list */
        seg_preclaim_cbfunc_t   p_callback;     /* reclaim callback function */
        clock_t                 p_lbolt;        /* lbolt from last use */
        struct seg_phash        *p_hashp;       /* our pcache hash bucket */
        uint_t                  p_active;       /* active count */
        uchar_t                 p_write;        /* true if S_WRITE */
        uchar_t                 p_ref;          /* reference byte */
        ushort_t                p_flags;        /* bit flags */
};

struct seg_phash {
        struct seg_pcache       *p_hnext;       /* list for hashed blocks */
        struct seg_pcache       *p_hprev;
        kmutex_t                p_hmutex;       /* protects hash bucket */
        pcache_link_t           p_halink[2];    /* active bucket linkages */
};

struct seg_phash_wired {
        struct seg_pcache       *p_hnext;       /* list for hashed blocks */
        struct seg_pcache       *p_hprev;
        kmutex_t                p_hmutex;       /* protects hash bucket */
};

/*
 * A parameter to control a maximum number of bytes that can be
 * purged from pcache at a time.
 */
#define P_MAX_APURGE_BYTES      (1024 * 1024 * 1024)

/*
 * log2(fraction of pcache to reclaim at a time).
 */
#define P_SHRINK_SHFT           (5)

/*
 * The following variables can be tuned via /etc/system.
 */

int     segpcache_enabled = 1;          /* if 1, shadow lists are cached */
pgcnt_t segpcache_maxwindow = 0;        /* max # of pages that can be cached */
ulong_t segpcache_hashsize_win = 0;     /* # of non wired buckets */
ulong_t segpcache_hashsize_wired = 0;   /* # of wired buckets */
int     segpcache_reap_sec = 1;         /* reap check rate in secs */
clock_t segpcache_reap_ticks = 0;       /* reap interval in ticks */
int     segpcache_pcp_maxage_sec = 1;   /* pcp max age in secs */
clock_t segpcache_pcp_maxage_ticks = 0; /* pcp max age in ticks */
int     segpcache_shrink_shift = P_SHRINK_SHFT; /* log2 reap fraction */
pgcnt_t segpcache_maxapurge_bytes = P_MAX_APURGE_BYTES; /* max purge bytes */

static kmutex_t seg_pcache_mtx; /* protects seg_pdisabled counter */
static kmutex_t seg_pasync_mtx; /* protects async thread scheduling */
static kcondvar_t seg_pasync_cv;

#pragma align 64(pctrl1)
#pragma align 64(pctrl2)
#pragma align 64(pctrl3)

/*
 * Keep frequently used variables together in one cache line.
 */
static struct p_ctrl1 {
        uint_t p_disabled;              /* if not 0, caching temporarily off */
        pgcnt_t p_maxwin;               /* max # of pages that can be cached */
        size_t p_hashwin_sz;            /* # of non wired buckets */
        struct seg_phash *p_htabwin;    /* hash table for non wired entries */
        size_t p_hashwired_sz;          /* # of wired buckets */
        struct seg_phash_wired *p_htabwired; /* hash table for wired entries */
        kmem_cache_t *p_kmcache;        /* kmem cache for seg_pcache structs */
#ifdef _LP64
        ulong_t pad[1];
#endif /* _LP64 */
} pctrl1;

static struct p_ctrl2 {
        kmutex_t p_mem_mtx;     /* protects window counter and p_halinks */
        pgcnt_t  p_locked_win;  /* # pages from window */
        pgcnt_t  p_locked;      /* # of pages cached by pagelock */
        uchar_t  p_ahcur;       /* current active links for insert/delete */
        uchar_t  p_athr_on;     /* async reclaim thread is running. */
        pcache_link_t p_ahhead[2]; /* active buckets linkages */
} pctrl2;

static struct p_ctrl3 {
        clock_t p_pcp_maxage;           /* max pcp age in ticks */
        ulong_t p_athr_empty_ahb;       /* athread walk stats */
        ulong_t p_athr_full_ahb;        /* athread walk stats */
        pgcnt_t p_maxpurge_npages;      /* max pages to purge at a time */
        int     p_shrink_shft;          /* reap shift factor */
#ifdef _LP64
        ulong_t pad[3];
#endif /* _LP64 */
} pctrl3;

#define seg_pdisabled                   pctrl1.p_disabled
#define seg_pmaxwindow                  pctrl1.p_maxwin
#define seg_phashsize_win               pctrl1.p_hashwin_sz
#define seg_phashtab_win                pctrl1.p_htabwin
#define seg_phashsize_wired             pctrl1.p_hashwired_sz
#define seg_phashtab_wired              pctrl1.p_htabwired
#define seg_pkmcache                    pctrl1.p_kmcache
#define seg_pmem_mtx                    pctrl2.p_mem_mtx
#define seg_plocked_window              pctrl2.p_locked_win
```

```
#define seg_plocked                     pctrl2.p_locked
#define seg_pahcur                      pctrl2.p_ahcur
#define seg_pathr_on                    pctrl2.p_athr_on
#define seg_pahhead                     pctrl2.p_ahhead
#define seg_pmax_pcpage                 pctrl3.p_pcp_maxage
#define seg_pathr_empty_ahb             pctrl3.p_athr_empty_ahb
#define seg_pathr_full_ahb              pctrl3.p_athr_full_ahb
#define seg_pshrink_shift               pctrl3.p_shrink_shft
#define seg_pmaxapurge_npages           pctrl3.p_maxapurge_npages

#define P_HASHWIN_MASK                  (seg_phashsize_win - 1)
#define P_HASHWIRED_MASK                (seg_phashsize_wired - 1)
#define P_BASESHIFT                     (6)

kthread_t *seg_pasync_thr;

extern struct seg_ops segvn_ops;
extern struct seg_ops segspt_shmops;

#define IS_PFLAGS_WIRED(flags) ((flags) & SEGP_FORCE_WIRED)
#define IS_PCP_WIRED(pcp) IS_PFLAGS_WIRED((pcp)->p_flags)

#define LBOLT_DELTA(t)  ((ulong_t)(ddi_get_lbolt() - (t)))

#define PCP_AGE(pcp)    LBOLT_DELTA((pcp)->p_lbolt)

/*
 * htag0 argument can be a seg or amp pointer.
 */
#define P_HASHBP(seg, htag0, addr, flags)                               \
        (IS_PFLAGS_WIRED((flags)) ?                                     \
            ((struct seg_phash *)&seg_phashtab_wired[P_HASHWIRED_MASK & \
            ((uintptr_t)(htag0) >> P_BASESHIFT)]) :                     \
            (&seg_phashtab_win[P_HASHWIN_MASK &                         \
            (((uintptr_t)(htag0) >> 3) ^                                \
            ((uintptr_t)(addr) >> ((flags & SEGP_PSHIFT) ?              \
            (flags >> 16) : page_get_shift((seg)->s_szc))))]))

/*
 * htag0 argument can be a seg or amp pointer.
 */
#define P_MATCH(pcp, htag0, addr, len)                                  \
        ((pcp)->p_htag0 == (htag0) &&                                   \
        (pcp)->p_addr == (addr) &&                                      \
        (pcp)->p_len >= (len))

#define P_MATCH_PP(pcp, htag0, addr, len, pp)                           \
        ((pcp)->p_pp == (pp) &&                                         \
        (pcp)->p_htag0 == (htag0) &&                                    \
        (pcp)->p_addr == (addr) &&                                      \
        (pcp)->p_len >= (len))

#define plink2pcache(pl)        ((struct seg_pcache *)((uintptr_t)(pl) - \
    offsetof(struct seg_pcache, p_plink)))

#define hlink2phash(hl, l)      ((struct seg_phash *)((uintptr_t)(hl) - \
    offsetof(struct seg_phash, p_halink[l])))

/*
 * seg_padd_abuck()/seg_premove_abuck() link and unlink hash buckets from
 * active hash bucket lists. We maintain active bucket lists to reduce the
 * overhead of finding active buckets during asynchronous purging since there
 * can be 10s of millions of buckets on a large system but only a small subset
 * of them in actual use.
 *
 * There’re 2 active bucket lists. Current active list (as per seg_pahcur) is
 * used by seg_pinsert()/seg_pinactive()/seg_ppurge() to add and delete
 * buckets. The other list is used by asynchronous purge thread. This allows
 * the purge thread to walk its active list without holding seg_pmem_mtx for a
 * long time. When asynchronous thread is done with its list it switches to
 * current active list and makes the list it just finished processing as
 * current active list.
 *
 * seg_padd_abuck() only adds the bucket to current list if the bucket is not
 * yet on any list.  seg_premove_abuck() may remove the bucket from either
 * list. If the bucket is on current list it will be always removed. Otherwise
 * the bucket is only removed if asynchronous purge thread is not currently
 * running or seg_premove_abuck() is called by asynchronous purge thread
 * itself. A given bucket can only be on one of active lists at a time. These
 * routines should be called with per bucket lock held.  The routines use
 * seg_pmem_mtx to protect list updates. seg_padd_abuck() must be called after
 * the first entry is added to the bucket chain and seg_premove_abuck() must
 * be called after the last pcp entry is deleted from its chain. Per bucket
 * lock should be held by the callers.  This avoids a potential race condition
 * when seg_premove_abuck() removes a bucket after pcp entries are added to
 * its list after the caller checked that the bucket has no entries. (this
 * race would cause a loss of an active bucket from the active lists).
 *
 * Both lists are circular doubly linked lists anchored at seg_pahhead heads.
 * New entries are added to the end of the list since LRU is used as the
 * purging policy.
 */
static void
seg_padd_abuck(struct seg_phash *hp)
{
        int lix;

        ASSERT(MUTEX_HELD(&hp->p_hmutex));
        ASSERT((struct seg_phash *)hp->p_hnext != hp);
        ASSERT((struct seg_phash *)hp->p_hprev != hp);
        ASSERT(hp->p_hnext == hp->p_hprev);
        ASSERT(!IS_PCP_WIRED(hp->p_hnext));
        ASSERT(hp->p_hnext->p_hnext == (struct seg_pcache *)hp);
        ASSERT(hp->p_hprev->p_hprev == (struct seg_pcache *)hp);
        ASSERT(hp >= seg_phashtab_win &&
            hp < &seg_phashtab_win[seg_phashsize_win]);

        /*
         * This bucket can already be on one of active lists
         * since seg_premove_abuck() may have failed to remove it
         * before.
         */
        mutex_enter(&seg_pmem_mtx);
        lix = seg_pahcur;
        ASSERT(lix >= 0 && lix <= 1);
        if (hp->p_halink[lix].p_lnext != NULL) {
                ASSERT(hp->p_halink[lix].p_lprev != NULL);
                ASSERT(hp->p_halink[!lix].p_lnext == NULL);
                ASSERT(hp->p_halink[!lix].p_lprev == NULL);
                mutex_exit(&seg_pmem_mtx);
                return;
        }
        ASSERT(hp->p_halink[lix].p_lprev == NULL);

        /*
         * If this bucket is still on list !lix async thread can't yet remove
         * it since we hold here per bucket lock. In this case just return
         * since async thread will eventually find and process this bucket.
         */
        if (hp->p_halink[!lix].p_lnext != NULL) {
                ASSERT(hp->p_halink[!lix].p_lprev != NULL);
                mutex_exit(&seg_pmem_mtx);
```

```
                return;
        }
        ASSERT(hp->p_halink[!lix].p_lprev == NULL);
        /*
         * This bucket is not on any active bucket list yet.
         * Add the bucket to the tail of current active list.
         */
        hp->p_halink[lix].p_lnext = &seg_pahhead[lix];
        hp->p_halink[lix].p_lprev = seg_pahhead[lix].p_lprev;
        seg_pahhead[lix].p_lprev->p_lnext = &hp->p_halink[lix];
        seg_pahhead[lix].p_lprev = &hp->p_halink[lix];
        mutex_exit(&seg_pmem_mtx);
}

static void
seg_premove_abuck(struct seg_phash *hp, int athr)
{
        int lix;

        ASSERT(MUTEX_HELD(&hp->p_hmutex));
        ASSERT((struct seg_phash *)hp->p_hnext == hp);
        ASSERT((struct seg_phash *)hp->p_hprev == hp);
        ASSERT(hp >= seg_phashtab_win &&
            hp < &seg_phashtab_win[seg_phashsize_win]);

        if (athr) {
                ASSERT(seg_pathr_on);
                ASSERT(seg_pahcur <= 1);
                /*
                 * We are called by asynchronous thread that found this bucket
                 * on not currently active (i.e. !seg_pahcur) list. Remove it
                 * from there.  Per bucket lock we are holding makes sure
                 * seg_pinsert() can't sneak in and add pcp entries to this
                 * bucket right before we remove the bucket from its list.
                 */
                lix = !seg_pahcur;
                ASSERT(hp->p_halink[lix].p_lnext != NULL);
                ASSERT(hp->p_halink[lix].p_lprev != NULL);
                ASSERT(hp->p_halink[!lix].p_lnext == NULL);
                ASSERT(hp->p_halink[!lix].p_lprev == NULL);
                hp->p_halink[lix].p_lnext->p_lprev = hp->p_halink[lix].p_lprev;
                hp->p_halink[lix].p_lprev->p_lnext = hp->p_halink[lix].p_lnext;
                hp->p_halink[lix].p_lnext = NULL;
                hp->p_halink[lix].p_lprev = NULL;
                return;
        }

        mutex_enter(&seg_pmem_mtx);
        lix = seg_pahcur;
        ASSERT(lix >= 0 && lix <= 1);

        /*
         * If the bucket is on currently active list just remove it from
         * there.
         */
        if (hp->p_halink[lix].p_lnext != NULL) {
                ASSERT(hp->p_halink[lix].p_lprev != NULL);
                ASSERT(hp->p_halink[!lix].p_lnext == NULL);
                ASSERT(hp->p_halink[!lix].p_lprev == NULL);
                hp->p_halink[lix].p_lnext->p_lprev = hp->p_halink[lix].p_lprev;
                hp->p_halink[lix].p_lprev->p_lnext = hp->p_halink[lix].p_lnext;
                hp->p_halink[lix].p_lnext = NULL;
                hp->p_halink[lix].p_lprev = NULL;
                mutex_exit(&seg_pmem_mtx);
                return;
        }
        ASSERT(hp->p_halink[lix].p_lprev == NULL);

        /*
         * If asynchronous thread is not running we can remove the bucket from
         * not currently active list. The bucket must be on this list since we
         * already checked that it’s not on the other list and the bucket from
         * which we just deleted the last pcp entry must be still on one of the
         * active bucket lists.
         */
        lix = !lix;
        ASSERT(hp->p_halink[lix].p_lnext != NULL);
        ASSERT(hp->p_halink[lix].p_lprev != NULL);

        if (!seg_pathr_on) {
                hp->p_halink[lix].p_lnext->p_lprev = hp->p_halink[lix].p_lprev;
                hp->p_halink[lix].p_lprev->p_lnext = hp->p_halink[lix].p_lnext;
                hp->p_halink[lix].p_lnext = NULL;
                hp->p_halink[lix].p_lprev = NULL;
        }
        mutex_exit(&seg_pmem_mtx);
}

/*
 * Check if bucket pointed by hp already has a pcp entry that matches request
 * htag0, addr and len. Set *found to 1 if match is found and to 0 otherwise.
 * Also delete matching entries that cover smaller address range but start
 * at the same address as addr argument. Return the list of deleted entries if
 * any. This is an internal helper function called from seg_pinsert() only
 * for non wired shadow lists. The caller already holds a per seg/amp list
 * lock.
 */
static struct seg_pcache *
seg_plookup_checkdup(struct seg_phash *hp, void *htag0,
    caddr_t addr, size_t len, int *found)
{
        struct seg_pcache *pcp;
        struct seg_pcache *delcallb_list = NULL;

        ASSERT(MUTEX_HELD(&hp->p_hmutex));

        *found = 0;
        for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
            pcp = pcp->p_hnext) {
                ASSERT(pcp->p_hashp == hp);
                if (pcp->p_htag0 == htag0 && pcp->p_addr == addr) {
                        ASSERT(!IS_PCP_WIRED(pcp));
                        if (pcp->p_len < len) {
                                pcache_link_t *plinkp;
                                if (pcp->p_active) {
                                        continue;
                                }
                                plinkp = &pcp->p_plink;
                                plinkp->p_lprev->p_lnext = plinkp->p_lnext;
                                plinkp->p_lnext->p_lprev = plinkp->p_lprev;
                                pcp->p_hprev->p_hnext = pcp->p_hnext;
                                pcp->p_hnext->p_hprev = pcp->p_hprev;
                                pcp->p_hprev = delcallb_list;
                                delcallb_list = pcp;
                        } else {
                                *found = 1;
                                break;
                        }
                }
        }
        return (delcallb_list);
}
```

```
/*
 * lookup an address range in pagelock cache. Return shadow list and bump up
 * active count. If amp is not NULL use amp as a lookup tag otherwise use seg
 * as a lookup tag.
 */
struct page **
seg_plookup(struct seg *seg, struct anon_map *amp, caddr_t addr, size_t len,
    enum seg_rw rw, uint_t flags)
{
        struct seg_pcache *pcp;
        struct seg_phash *hp;
        void *htag0;

        ASSERT(seg != NULL);
        ASSERT(rw == S_READ || rw == S_WRITE);

        /*
         * Skip pagelock cache, while DR is in progress or
         * seg_pcache is off.
         */
        if (seg_pdisabled) {
                return (NULL);
        }
        ASSERT(seg_phashsize_win != 0);

        htag0 = (amp == NULL ? (void *)seg : (void *)amp);
        hp = P_HASHBP(seg, htag0, addr, flags);
        mutex_enter(&hp->p_hmutex);
        for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
            pcp = pcp->p_hnext) {
                ASSERT(pcp->p_hashp == hp);
                if (P_MATCH(pcp, htag0, addr, len)) {
                        ASSERT(IS_PFLAGS_WIRED(flags) == IS_PCP_WIRED(pcp));
                        /*
                         * If this request wants to write pages
                         * but write permissions starting from
                         * addr don’t cover the entire length len
                         * return lookup failure back to the caller.
                         * It will check protections and fail this
                         * pagelock operation with EACCESS error.
                         */
                        if (rw == S_WRITE && pcp->p_wlen < len) {
                                break;
                        }
                        if (pcp->p_active == UINT_MAX) {
                                break;
                        }
                        pcp->p_active++;
                        if (rw == S_WRITE && !pcp->p_write) {
                                pcp->p_write = 1;
                        }
                        mutex_exit(&hp->p_hmutex);
                        return (pcp->p_pp);
                }
        }
        mutex_exit(&hp->p_hmutex);
        return (NULL);
}

/*
 * mark address range inactive. If the cache is off or the address range is
 * not in the cache or another shadow list that covers bigger range is found
 * we call the segment driver to reclaim the pages. Otherwise just decrement
 * active count and set ref bit.  If amp is not NULL use amp as a lookup tag
 * otherwise use seg as a lookup tag.
 */
void
seg_pinactive(struct seg *seg, struct anon_map *amp, caddr_t addr,
    size_t len, struct page **pp, enum seg_rw rw, uint_t flags,
    seg_preclaim_cbfunc_t callback)
{
        struct seg_pcache *pcp;
        struct seg_phash *hp;
        kmutex_t *pmtx = NULL;
        pcache_link_t *pheadp;
        void *htag0;
        pgcnt_t npages = 0;
        int keep = 0;

        ASSERT(seg != NULL);
        ASSERT(rw == S_READ || rw == S_WRITE);

        htag0 = (amp == NULL ? (void *)seg : (void *)amp);

        /*
         * Skip lookup if pcache is not configured.
         */
        if (seg_phashsize_win == 0) {
                goto out;
        }

        /*
         * Grab per seg/amp lock before hash lock if we are going to remove
         * inactive entry from pcache.
         */
        if (!IS_PFLAGS_WIRED(flags) && seg_pdisabled) {
                if (amp == NULL) {
                        pheadp = &seg->s_phead;
                        pmtx = &seg->s_pmtx;
                } else {
                        pheadp = &amp->a_phead;
                        pmtx = &amp->a_pmtx;
                }
                mutex_enter(pmtx);
        }

        hp = P_HASHBP(seg, htag0, addr, flags);
        mutex_enter(&hp->p_hmutex);
again:
        for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
            pcp = pcp->p_hnext) {
                ASSERT(pcp->p_hashp == hp);
                if (P_MATCH_PP(pcp, htag0, addr, len, pp)) {
                        ASSERT(IS_PFLAGS_WIRED(flags) == IS_PCP_WIRED(pcp));
                        ASSERT(pcp->p_active);
                        if (keep) {
                                /*
                                 * Don’t remove this pcp entry
                                 * if we didn’t find duplicate
                                 * shadow lists on second search.
                                 * Somebody removed those duplicates
                                 * since we dropped hash lock after first
                                 * search.
                                 */
                                ASSERT(pmtx != NULL);
                                ASSERT(!IS_PFLAGS_WIRED(flags));
                                mutex_exit(pmtx);
                                pmtx = NULL;
                        }
                        pcp->p_active--;
                        if (pcp->p_active == 0 && (pmtx != NULL ||
```

```
                            (seg_pdisabled && IS_PFLAGS_WIRED(flags)))) {

                                /*
                                 * This entry is no longer active.  Remove it
                                 * now either because pcaching is temporarily
                                 * disabled or there’re other pcp entries that
                                 * can match this pagelock request (i.e. this
                                 * entry is a duplicate).
                                 */

                                ASSERT(callback == pcp->p_callback);
                                if (pmtx != NULL) {
                                        pcache_link_t *plinkp = &pcp->p_plink;
                                        ASSERT(!IS_PCP_WIRED(pcp));
                                        ASSERT(pheadp->p_lnext != pheadp);
                                        ASSERT(pheadp->p_lprev != pheadp);
                                        plinkp->p_lprev->p_lnext =
                                            plinkp->p_lnext;
                                        plinkp->p_lnext->p_lprev =
                                            plinkp->p_lprev;
                                }
                                pcp->p_hprev->p_hnext = pcp->p_hnext;
                                pcp->p_hnext->p_hprev = pcp->p_hprev;
                                if (!IS_PCP_WIRED(pcp) &&
                                    hp->p_hnext == (struct seg_pcache *)hp) {
                                        /*
                                         * We removed the last entry from this
                                         * bucket.  Now remove the bucket from
                                         * its active list.
                                         */
                                        seg_premove_abuck(hp, 0);
                                }
                                mutex_exit(&hp->p_hmutex);
                                if (pmtx != NULL) {
                                        mutex_exit(pmtx);
                                }
                                len = pcp->p_len;
                                npages = btop(len);
                                if (rw != S_WRITE && pcp->p_write) {
                                        rw = S_WRITE;
                                }
                                kmem_cache_free(seg_pkmcache, pcp);
                                goto out;
                        } else {
                                /*
                                 * We found a matching pcp entry but will not
                                 * free it right away even if it’s no longer
                                 * active.
                                 */
                                if (!pcp->p_active && !IS_PCP_WIRED(pcp)) {
                                        /*
                                         * Set the reference bit and mark the
                                         * time of last access to this pcp
                                         * so that asynchronous thread doesn’t
                                         * free it immediately since
                                         * it may be reactivated very soon.
                                         */
                                        pcp->p_lbolt = ddi_get_lbolt();
                                        pcp->p_ref = 1;
                                }
                                mutex_exit(&hp->p_hmutex);
                                if (pmtx != NULL) {
                                        mutex_exit(pmtx);
                                }
                                return;
                        }
```

```
                } else if (!IS_PFLAGS_WIRED(flags) &&
                    P_MATCH(pcp, htag0, addr, len)) {
                        /*
                         * This is a duplicate pcp entry.  This situation may
                         * happen if a bigger shadow list that covers our
                         * range was added while our entry was still active.
                         * Now we can free our pcp entry if it becomes
                         * inactive.
                         */
                        if (!pcp->p_active) {
                                /*
                                 * Mark this entry as referenced just in case
                                 * we'll free our own pcp entry soon.
                                 */
                                pcp->p_lbolt = ddi_get_lbolt();
                                pcp->p_ref = 1;
                        }
                        if (pmtx != NULL) {
                                /*
                                 * we are already holding pmtx and found a
                                 * duplicate.  Don't keep our own pcp entry.
                                 */
                                keep = 0;
                                continue;
                        }
                        /*
                         * We have to use mutex_tryenter to attempt to lock
                         * seg/amp list lock since we already hold hash lock
                         * and seg/amp list lock is above hash lock in lock
                         * order.  If mutex_tryenter fails drop hash lock and
                         * retake both locks in correct order and research
                         * this hash chain.
                         */
                        ASSERT(keep == 0);
                        if (amp == NULL) {
                                pheadp = &seg->s_phead;
                                pmtx = &seg->s_pmtx;
                        } else {
                                pheadp = &amp->a_phead;
                                pmtx = &amp->a_pmtx;
                        }
                        if (!mutex_tryenter(pmtx)) {
                                mutex_exit(&hp->p_hmutex);
                                mutex_enter(pmtx);
                                mutex_enter(&hp->p_hmutex);
                                /*
                                 * If we don't find bigger shadow list on
                                 * second search (it may happen since we
                                 * dropped bucket lock) keep the entry that
                                 * matches our own shadow list.
                                 */
                                keep = 1;
                                goto again;
                        }
                }
        }
        mutex_exit(&hp->p_hmutex);
        if (pmtx != NULL) {
                mutex_exit(pmtx);
        }
out:
        (*callback)(htag0, addr, len, pp, rw, 0);
        if (npages) {
                mutex_enter(&seg_pmem_mtx);
                ASSERT(seg_plocked >= npages);
                seg_plocked -= npages;
```

```
                if (!IS_PFLAGS_WIRED(flags)) {
                        ASSERT(seg_plocked_window >= npages);
                        seg_plocked_window -= npages;
                }
                mutex_exit(&seg_pmem_mtx);
        }

}

#ifdef DEBUG
static uint32_t p_insert_chk_mtbf = 0;
#endif

/*
 * The seg_pinsert_check() is used by segment drivers to predict whether
 * a call to seg_pinsert will fail and thereby avoid wasteful pre-processing.
 */
/*ARGSUSED*/
int
seg_pinsert_check(struct seg *seg, struct anon_map *amp, caddr_t addr,
    size_t len, uint_t flags)
{
        ASSERT(seg != NULL);

#ifdef DEBUG
        if (p_insert_chk_mtbf && !(gethrtime() % p_insert_chk_mtbf)) {
                return (SEGP_FAIL);
        }
#endif

        if (seg_pdisabled) {
                return (SEGP_FAIL);
        }
        ASSERT(seg_phashsize_win != 0);

        if (IS_PFLAGS_WIRED(flags)) {
                return (SEGP_SUCCESS);
        }

        if (seg_plocked_window + btop(len) > seg_pmaxwindow) {
                return (SEGP_FAIL);
        }

        if (freemem < desfree) {
                return (SEGP_FAIL);
        }

        return (SEGP_SUCCESS);
}

#ifdef DEBUG
static uint32_t p_insert_mtbf = 0;
#endif

/*
 * Insert address range with shadow list into pagelock cache if there's no
 * shadow list already cached for this address range. If the cache is off or
 * caching is temporarily disabled or the allowed 'window' is exceeded return
 * SEGP_FAIL. Otherwise return SEGP_SUCCESS.
 *
 * For non wired shadow lists (segvn case) include address in the hashing
 * function to avoid linking all the entries from the same segment or amp on
 * the same bucket.  amp is used instead of seg if amp is not NULL. Non wired
 * pcache entries are also linked on a per segment/amp list so that all
 * entries can be found quickly during seg/amp purge without walking the
 * entire pcache hash table.  For wired shadow lists (segspt case) we
 * don’t use address hashing and per segment linking because the caller
 * currently inserts only one entry per segment that covers the entire
 * segment. If we used per segment linking even for segspt it would complicate
 * seg_ppurge_wiredpp() locking.
 *
 * Both hash bucket and per seg/amp locks need to be held before adding a non
 * wired entry to hash and per seg/amp lists. per seg/amp lock should be taken
 * first.
 *
 * This function will also remove from pcache old inactive shadow lists that
 * overlap with this request but cover smaller range for the same start
 * address.
 */
int
seg_pinsert(struct seg *seg, struct anon_map *amp, caddr_t addr, size_t len,
    size_t wlen, struct page **pp, enum seg_rw rw, uint_t flags,
    seg_preclaim_cbfunc_t callback)
{
        struct seg_pcache *pcp;
        struct seg_phash *hp;
        pgcnt_t npages;
        pcache_link_t *pheadp;
        kmutex_t *pmtx;
        struct seg_pcache *delcallb_list = NULL;

        ASSERT(seg != NULL);
        ASSERT(rw == S_READ || rw == S_WRITE);
        ASSERT(rw == S_READ || wlen == len);
        ASSERT(rw == S_WRITE || wlen <= len);
        ASSERT(amp == NULL || wlen == len);

#ifdef DEBUG
        if (p_insert_mtbf && !(gethrtime() % p_insert_mtbf)) {
                return (SEGP_FAIL);
        }
#endif

        if (seg_pdisabled) {
                return (SEGP_FAIL);
        }
        ASSERT(seg_phashsize_win != 0);

        ASSERT((len & PAGEOFFSET) == 0);
        npages = btop(len);
        mutex_enter(&seg_pmem_mtx);
        if (!IS_PFLAGS_WIRED(flags)) {
                if (seg_plocked_window + npages > seg_pmaxwindow) {
                        mutex_exit(&seg_pmem_mtx);
                        return (SEGP_FAIL);
                }
                seg_plocked_window += npages;
        }
        seg_plocked += npages;
        mutex_exit(&seg_pmem_mtx);

        pcp = kmem_cache_alloc(seg_pkmcache, KM_SLEEP);
        /*
         * If amp is not NULL set htag0 to amp otherwise set it to seg.
         */
        if (amp == NULL) {
                pcp->p_htag0 = (void *)seg;
                pcp->p_flags = flags & 0xffff;
        } else {
                pcp->p_htag0 = (void *)amp;
                pcp->p_flags = (flags & 0xffff) | SEGP_AMP;
        }
```

```
        pcp->p_addr = addr;
        pcp->p_len = len;
        pcp->p_wlen = wlen;
        pcp->p_pp = pp;
        pcp->p_write = (rw == S_WRITE);
        pcp->p_callback = callback;
        pcp->p_active = 1;

        hp = P_HASHBP(seg, pcp->p_htag0, addr, flags);
        if (!IS_PFLAGS_WIRED(flags)) {
                int found;
                void *htag0;
                if (amp == NULL) {
                        pheadp = &seg->s_phead;
                        pmtx = &seg->s_pmtx;
                        htag0 = (void *)seg;
                } else {
                        pheadp = &amp->a_phead;
                        pmtx = &amp->a_pmtx;
                        htag0 = (void *)amp;
                }
                mutex_enter(pmtx);
                mutex_enter(&hp->p_hmutex);
                delcallb_list = seg_plookup_checkdup(hp, htag0, addr,
                    len, &found);
                if (found) {
                        mutex_exit(&hp->p_hmutex);
                        mutex_exit(pmtx);
                        mutex_enter(&seg_pmem_mtx);
                        seg_plocked -= npages;
                        seg_plocked_window -= npages;
                        mutex_exit(&seg_pmem_mtx);
                        kmem_cache_free(seg_pkmcache, pcp);
                        goto out;
                }
                pcp->p_plink.p_lnext = pheadp->p_lnext;
                pcp->p_plink.p_lprev = pheadp;
                pheadp->p_lnext->p_lprev = &pcp->p_plink;
                pheadp->p_lnext = &pcp->p_plink;
        } else {
                mutex_enter(&hp->p_hmutex);
        }
        pcp->p_hashp = hp;
        pcp->p_hnext = hp->p_hnext;
        pcp->p_hprev = (struct seg_pcache *)hp;
        hp->p_hnext->p_hprev = pcp;
        hp->p_hnext = pcp;
        if (!IS_PFLAGS_WIRED(flags) &&
            hp->p_hprev == pcp) {
                seg_padd_abuck(hp);
        }
        mutex_exit(&hp->p_hmutex);
        if (!IS_PFLAGS_WIRED(flags)) {
                mutex_exit(pmtx);
        }

out:
        npages = 0;
        while (delcallb_list != NULL) {
                pcp = delcallb_list;
                delcallb_list = pcp->p_hprev;
                ASSERT(!IS_PCP_WIRED(pcp) && !pcp->p_active);
                (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
                    pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 0);
                npages += btop(pcp->p_len);
                kmem_cache_free(seg_pkmcache, pcp);
        }
        if (npages) {
                ASSERT(!IS_PFLAGS_WIRED(flags));
                mutex_enter(&seg_pmem_mtx);
                ASSERT(seg_plocked >= npages);
                ASSERT(seg_plocked_window >= npages);
                seg_plocked -= npages;
                seg_plocked_window -= npages;
                mutex_exit(&seg_pmem_mtx);
        }

        return (SEGP_SUCCESS);
}

/*
 * purge entries from the pagelock cache if not active
 * and not recently used.
 */
static void
seg_ppurge_async(int force)
{
        struct seg_pcache *delcallb_list = NULL;
        struct seg_pcache *pcp;
        struct seg_phash *hp;
        pgcnt_t npages = 0;
        pgcnt_t npages_window = 0;
        pgcnt_t npgs_to_purge;
        pgcnt_t npgs_purged = 0;
        int hlinks = 0;
        int hlix;
        pcache_link_t *hlinkp;
        pcache_link_t *hlnextp = NULL;
        int lowmem;
        int trim;

        ASSERT(seg_phashsize_win != 0);

        /*
         * if the cache is off or empty, return
         */
        if (seg_plocked == 0 || (!force && seg_plocked_window == 0)) {
                return;
        }

        if (!force) {
                lowmem = 0;
                trim = 0;
                if (freemem < lotsfree + needfree) {
                        spgcnt_t fmem = MAX((spgcnt_t)(freemem - needfree), 0);
                        if (fmem <= 5 * (desfree >> 2)) {
                                lowmem = 1;
                        } else if (fmem <= 7 * (lotsfree >> 3)) {
                                if (seg_plocked_window >=
                                    (availrmem_initial >> 1)) {
                                        lowmem = 1;
                                }
                        } else if (fmem < lotsfree) {
                                if (seg_plocked_window >=
                                    3 * (availrmem_initial >> 2)) {
                                        lowmem = 1;
                                }
                        }
                }
                if (seg_plocked_window >= 7 * (seg_pmaxwindow >> 3)) {
                        trim = 1;
                }
```

```
                if (!lowmem && !trim) {
                        return;
                }
                npgs_to_purge = seg_plocked_window >>
                    seg_pshrink_shift;
                if (lowmem) {
                        npgs_to_purge = MIN(npgs_to_purge,
                            MAX(seg_pmaxapurge_npages, desfree));
                } else {
                        npgs_to_purge = MIN(npgs_to_purge,
                            seg_pmaxapurge_npages);
                }
                if (npgs_to_purge == 0) {
                        return;
                }
        } else {
                struct seg_phash_wired *hpw;

                ASSERT(seg_phashsize_wired != 0);

                for (hpw = seg_phashtab_wired;
                    hpw < &seg_phashtab_wired[seg_phashsize_wired]; hpw++) {

                        if (hpw->p_hnext == (struct seg_pcache *)hpw) {
                                continue;
                        }

                        mutex_enter(&hpw->p_hmutex);

                        for (pcp = hpw->p_hnext;
                            pcp != (struct seg_pcache *)hpw;
                            pcp = pcp->p_hnext) {

                                ASSERT(IS_PCP_WIRED(pcp));
                                ASSERT(pcp->p_hashp ==
                                    (struct seg_phash *)hpw);

                                if (pcp->p_active) {
                                        continue;
                                }
                                pcp->p_hprev->p_hnext = pcp->p_hnext;
                                pcp->p_hnext->p_hprev = pcp->p_hprev;
                                pcp->p_hprev = delcallb_list;
                                delcallb_list = pcp;
                        }
                        mutex_exit(&hpw->p_hmutex);
                }
        }

        mutex_enter(&seg_pmem_mtx);
        if (seg_pathr_on) {
                mutex_exit(&seg_pmem_mtx);
                goto runcb;
        }
        seg_pathr_on = 1;
        mutex_exit(&seg_pmem_mtx);
        ASSERT(seg_pahcur <= 1);
        hlix = !seg_pahcur;

again:
        for (hlinkp = seg_pahhead[hlix].p_lnext; hlinkp != &seg_pahhead[hlix];
            hlinkp = hlnextp) {

                hlnextp = hlinkp->p_lnext;
                ASSERT(hlnextp != NULL);
                hp = hlink2phash(hlinkp, hlix);
                if (hp->p_hnext == (struct seg_pcache *)hp) {
                        seg_pathr_empty_ahb++;
                        continue;
                }
                seg_pathr_full_ahb++;
                mutex_enter(&hp->p_hmutex);

                for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
                    pcp = pcp->p_hnext) {
                        pcache_link_t *pheadp;
                        pcache_link_t *plinkp;
                        void *htag0;
                        kmutex_t *pmtx;

                        ASSERT(!IS_PCP_WIRED(pcp));
                        ASSERT(pcp->p_hashp == hp);

                        if (pcp->p_active) {
                                continue;
                        }
                        if (!force && pcp->p_ref &&
                            PCP_AGE(pcp) < seg_pmax_pcpage) {
                                pcp->p_ref = 0;
                                continue;
                        }
                        plinkp = &pcp->p_plink;
                        htag0 = pcp->p_htag0;
                        if (pcp->p_flags & SEGP_AMP) {
                                pheadp = &((amp_t *)htag0)->a_phead;
                                pmtx = &((amp_t *)htag0)->a_pmtx;
                        } else {
                                pheadp = &((seg_t *)htag0)->s_phead;
                                pmtx = &((seg_t *)htag0)->s_pmtx;
                        }
                        if (!mutex_tryenter(pmtx)) {
                                continue;
                        }
                        ASSERT(pheadp->p_lnext != pheadp);
                        ASSERT(pheadp->p_lprev != pheadp);
                        plinkp->p_lprev->p_lnext =
                            plinkp->p_lnext;
                        plinkp->p_lnext->p_lprev =
                            plinkp->p_lprev;
                        pcp->p_hprev->p_hnext = pcp->p_hnext;
                        pcp->p_hnext->p_hprev = pcp->p_hprev;
                        mutex_exit(pmtx);
                        pcp->p_hprev = delcallb_list;
                        delcallb_list = pcp;
                        npgs_purged += btop(pcp->p_len);
                }
                if (hp->p_hnext == (struct seg_pcache *)hp) {
                        seg_premove_abuck(hp, 1);
                }
                mutex_exit(&hp->p_hmutex);
                if (npgs_purged >= seg_plocked_window) {
                        break;
                }
                if (!force) {
                        if (npgs_purged >= npgs_to_purge) {
                                break;
                        }
                        if (!trim && !(seg_pathr_full_ahb & 15)) {
                                ASSERT(lowmem);
                                if (freemem >= lotsfree + needfree) {
                                        break;
```

```
                                }
                        }
                }
        }

        if (hlinkp == &seg_pahhead[hlix]) {
                /*
                 * We processed the entire hlix active bucket list
                 * but didn’t find enough pages to reclaim.
                 * Switch the lists and walk the other list
                 * if we haven’t done it yet.
                 */
                mutex_enter(&seg_pmem_mtx);
                ASSERT(seg_pathr_on);
                ASSERT(seg_pahcur == !hlix);
                seg_pahcur = hlix;
                mutex_exit(&seg_pmem_mtx);
                if (++hlinks < 2) {
                        hlix = !hlix;
                        goto again;
                }
        } else if ((hlinkp = hlnextp) != &seg_pahhead[hlix] &&
            seg_pahhead[hlix].p_lnext != hlinkp) {
                ASSERT(hlinkp != NULL);
                ASSERT(hlinkp->p_lprev != &seg_pahhead[hlix]);
                ASSERT(seg_pahhead[hlix].p_lnext != &seg_pahhead[hlix]);
                ASSERT(seg_pahhead[hlix].p_lprev != &seg_pahhead[hlix]);

                /*
                 * Reinsert the header to point to hlinkp
                 * so that we start from hlinkp bucket next time around.
                 */
                seg_pahhead[hlix].p_lnext->p_lprev = seg_pahhead[hlix].p_lprev;
                seg_pahhead[hlix].p_lprev->p_lnext = seg_pahhead[hlix].p_lnext;
                seg_pahhead[hlix].p_lnext = hlinkp;
                seg_pahhead[hlix].p_lprev = hlinkp->p_lprev;
                hlinkp->p_lprev->p_lnext = &seg_pahhead[hlix];
                hlinkp->p_lprev = &seg_pahhead[hlix];
        }

        mutex_enter(&seg_pmem_mtx);
        ASSERT(seg_pathr_on);
        seg_pathr_on = 0;
        mutex_exit(&seg_pmem_mtx);

runcb:
        /*
         * Run the delayed callback list. segments/amps can’t go away until
         * callback is executed since they must have non 0 softlockcnt. That’s
         * why we don’t need to hold as/seg/amp locks to execute the callback.
         */
        while (delcallb_list != NULL) {
                pcp = delcallb_list;
                delcallb_list = pcp->p_hprev;
                ASSERT(!pcp->p_active);
                (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
                    pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 1);
                npages += btop(pcp->p_len);
                if (!IS_PCP_WIRED(pcp)) {
                        npages_window += btop(pcp->p_len);
                }
                kmem_cache_free(seg_pkmcache, pcp);
        }
        if (npages) {
                mutex_enter(&seg_pmem_mtx);
                ASSERT(seg_plocked >= npages);
                ASSERT(seg_plocked_window >= npages_window);
                seg_plocked -= npages;
                seg_plocked_window -= npages_window;
                mutex_exit(&seg_pmem_mtx);
        }
}

/*
 * Remove cached pages for segment(s) entries from hashtable.  The segments
 * are identified by pp array. This is useful for multiple seg's cached on
 * behalf of dummy segment (ISM/DISM) with common pp array.
 */
void
seg_ppurge_wiredpp(struct page **pp)
{
        struct seg_pcache *pcp;
        struct seg_phash_wired *hp;
        pgcnt_t npages = 0;
        struct  seg_pcache *delcallb_list = NULL;

        /*
         * if the cache is empty, return
         */
        if (seg_plocked == 0) {
                return;
        }
        ASSERT(seg_phashsize_wired != 0);

        for (hp = seg_phashtab_wired;
            hp < &seg_phashtab_wired[seg_phashsize_wired]; hp++) {
                if (hp->p_hnext == (struct seg_pcache *)hp) {
                        continue;
                }
                mutex_enter(&hp->p_hmutex);
                pcp = hp->p_hnext;
                while (pcp != (struct seg_pcache *)hp) {
                        ASSERT(pcp->p_hashp == (struct seg_phash *)hp);
                        ASSERT(IS_PCP_WIRED(pcp));
                        /*
                         * purge entries which are not active
                         */
                        if (!pcp->p_active && pcp->p_pp == pp) {
                                ASSERT(pcp->p_htag0 != NULL);
                                pcp->p_hprev->p_hnext = pcp->p_hnext;
                                pcp->p_hnext->p_hprev = pcp->p_hprev;
                                pcp->p_hprev = delcallb_list;
                                delcallb_list = pcp;
                        }
                        pcp = pcp->p_hnext;
                }
                mutex_exit(&hp->p_hmutex);
                /*
                 * segments can't go away until callback is executed since
                 * they must have non 0 softlockcnt. That's why we don't
                 * need to hold as/seg locks to execute the callback.
                 */
                while (delcallb_list != NULL) {
                        int done;
                        pcp = delcallb_list;
                        delcallb_list = pcp->p_hprev;
                        ASSERT(!pcp->p_active);
                        done = (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
                            pcp->p_len, pcp->p_pp,
                            pcp->p_write ? S_WRITE : S_READ, 1);
                        npages += btop(pcp->p_len);
                        ASSERT(IS_PCP_WIRED(pcp));
```

```
                        kmem_cache_free(seg_pkmcache, pcp);
                        if (done) {
                                ASSERT(delcallb_list == NULL);
                                goto out;
                        }
                }
        }

out:
        mutex_enter(&seg_pmem_mtx);
        ASSERT(seg_plocked >= npages);
        seg_plocked -= npages;
        mutex_exit(&seg_pmem_mtx);
}

/*
 * purge all entries for a given segment. Since we
 * callback into the segment driver directly for page
 * reclaim the caller needs to hold the right locks.
 */
void
seg_ppurge(struct seg *seg, struct anon_map *amp, uint_t flags)
{
        struct seg_pcache *delcallb_list = NULL;
        struct seg_pcache *pcp;
        struct seg_phash *hp;
        pgcnt_t npages = 0;
        void *htag0;

        if (seg_plocked == 0) {
                return;
        }
        ASSERT(seg_phashsize_win != 0);

        /*
         * If amp is not NULL use amp as a lookup tag otherwise use seg
         * as a lookup tag.
         */
        htag0 = (amp == NULL ? (void *)seg : (void *)amp);
        ASSERT(htag0 != NULL);
        if (IS_PFLAGS_WIRED(flags)) {
                hp = P_HASHBP(seg, htag0, 0, flags);
                mutex_enter(&hp->p_hmutex);
                pcp = hp->p_hnext;
                while (pcp != (struct seg_pcache *)hp) {
                        ASSERT(pcp->p_hashp == hp);
                        ASSERT(IS_PCP_WIRED(pcp));
                        if (pcp->p_htag0 == htag0) {
                                if (pcp->p_active) {
                                        break;
                                }
                                pcp->p_hprev->p_hnext = pcp->p_hnext;
                                pcp->p_hnext->p_hprev = pcp->p_hprev;
                                pcp->p_hprev = delcallb_list;
                                delcallb_list = pcp;
                        }
                        pcp = pcp->p_hnext;
                }
                mutex_exit(&hp->p_hmutex);
        } else {
                pcache_link_t *plinkp;
                pcache_link_t *pheadp;
                kmutex_t *pmtx;

                if (amp == NULL) {
                        ASSERT(seg != NULL);
                        pheadp = &seg->s_phead;
                        pmtx = &seg->s_pmtx;
                } else {
                        pheadp = &amp->a_phead;
                        pmtx = &amp->a_pmtx;
                }
                mutex_enter(pmtx);
                while ((plinkp = pheadp->p_lnext) != pheadp) {
                        pcp = plink2pcache(plinkp);
                        ASSERT(!IS_PCP_WIRED(pcp));
                        ASSERT(pcp->p_htag0 == htag0);
                        hp = pcp->p_hashp;
                        mutex_enter(&hp->p_hmutex);
                        if (pcp->p_active) {
                                mutex_exit(&hp->p_hmutex);
                                break;
                        }
                        ASSERT(plinkp->p_lprev == pheadp);
                        pheadp->p_lnext = plinkp->p_lnext;
                        plinkp->p_lnext->p_lprev = pheadp;
                        pcp->p_hprev->p_hnext = pcp->p_hnext;
                        pcp->p_hnext->p_hprev = pcp->p_hprev;
                        pcp->p_hprev = delcallb_list;
                        delcallb_list = pcp;
                        if (hp->p_hnext == (struct seg_pcache *)hp) {
                                seg_premove_abuck(hp, 0);
                        }
                        mutex_exit(&hp->p_hmutex);
                }
                mutex_exit(pmtx);
        }
        while (delcallb_list != NULL) {
                pcp = delcallb_list;
                delcallb_list = pcp->p_hprev;
                ASSERT(!pcp->p_active);
                (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr, pcp->p_len,
                    pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 0);
                npages += btop(pcp->p_len);
                kmem_cache_free(seg_pkmcache, pcp);
        }
        mutex_enter(&seg_pmem_mtx);
        ASSERT(seg_plocked >= npages);
        seg_plocked -= npages;
        if (!IS_PFLAGS_WIRED(flags)) {
                ASSERT(seg_plocked_window >= npages);
                seg_plocked_window -= npages;
        }
        mutex_exit(&seg_pmem_mtx);
}

static void seg_pinit_mem_config(void);

/*
 * setup the pagelock cache
 */
static void
seg_pinit(void)
{
        struct seg_phash *hp;
        ulong_t i;
        pgcnt_t physmegs;

        seg_plocked = 0;
        seg_plocked_window = 0;

        if (segpcache_enabled == 0) {
```

```
                seg_phashsize_win = 0;
                seg_phashsize_wired = 0;
                seg_pdisabled = 1;
                return;
        }

        seg_pdisabled = 0;
        seg_pkmcache = kmem_cache_create("seg_pcache",
            sizeof (struct seg_pcache), 0, NULL, NULL, NULL, NULL, NULL, 0);
        if (segpcache_pcp_maxage_ticks <= 0) {
                segpcache_pcp_maxage_ticks = segpcache_pcp_maxage_sec * hz;
        }
        seg_pmax_pcpage = segpcache_pcp_maxage_ticks;
        seg_pathr_empty_ahb = 0;
        seg_pathr_full_ahb = 0;
        seg_pshrink_shift = segpcache_shrink_shift;
        seg_pmaxapurge_npages = btop(segpcache_maxapurge_bytes);

        mutex_init(&seg_pcache_mtx, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&seg_pmem_mtx, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&seg_pasync_mtx, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&seg_pasync_cv, NULL, CV_DEFAULT, NULL);

        physmegs = physmem >> (20 - PAGESHIFT);

        /*
         * If segpcache_hashsize_win was not set in /etc/system or it has
         * absurd value set it to a default.
         */
        if (segpcache_hashsize_win == 0 || segpcache_hashsize_win > physmem) {
                /*
                 * Create one bucket per 32K (or at least per 8 pages) of
                 * available memory.
                 */
                pgcnt_t pages_per_bucket = MAX(btop(32 * 1024), 8);
                segpcache_hashsize_win = MAX(1024, physmem / pages_per_bucket);
        }
        if (!ISP2(segpcache_hashsize_win)) {
                ulong_t rndfac = ~(1UL <<
                    (highbit(segpcache_hashsize_win) - 1));
                rndfac &= segpcache_hashsize_win;
                segpcache_hashsize_win += rndfac;
                segpcache_hashsize_win = 1 <<
                    (highbit(segpcache_hashsize_win) - 1);
        }
        seg_phashsize_win = segpcache_hashsize_win;
        seg_phashtab_win = kmem_zalloc(
            seg_phashsize_win * sizeof (struct seg_phash),
            KM_SLEEP);
        for (i = 0; i < seg_phashsize_win; i++) {
                hp = &seg_phashtab_win[i];
                hp->p_hnext = (struct seg_pcache *)hp;
                hp->p_hprev = (struct seg_pcache *)hp;
                mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
        }

        seg_pahcur = 0;
        seg_pathr_on = 0;
        seg_pahhead[0].p_lnext = &seg_pahhead[0];
        seg_pahhead[0].p_lprev = &seg_pahhead[0];
        seg_pahhead[1].p_lnext = &seg_pahhead[1];
        seg_pahhead[1].p_lprev = &seg_pahhead[1];

        /*
         * If segpcache_hashsize_wired was not set in /etc/system or it has
         * absurd value set it to a default.
         */
        if (segpcache_hashsize_wired == 0 ||
            segpcache_hashsize_wired > physmem / 4) {
                /*
                 * Choose segpcache_hashsize_wired based on physmem.
                 * Create a bucket per 128K bytes upto 256K buckets.
                 */
                if (physmegs < 20 * 1024) {
                        segpcache_hashsize_wired = MAX(1024, physmegs << 3);
                } else {
                        segpcache_hashsize_wired = 256 * 1024;
                }
        }
        if (!ISP2(segpcache_hashsize_wired)) {
                segpcache_hashsize_wired = 1 <<
                    highbit(segpcache_hashsize_wired);
        }
        seg_phashsize_wired = segpcache_hashsize_wired;
        seg_phashtab_wired = kmem_zalloc(
            seg_phashsize_wired * sizeof (struct seg_phash_wired), KM_SLEEP);
        for (i = 0; i < seg_phashsize_wired; i++) {
                hp = (struct seg_phash *)&seg_phashtab_wired[i];
                hp->p_hnext = (struct seg_pcache *)hp;
                hp->p_hprev = (struct seg_pcache *)hp;
                mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
        }

        if (segpcache_maxwindow == 0) {
                if (physmegs < 64) {
                        /* 3% of memory */
                        segpcache_maxwindow = availrmem >> 5;
                } else if (physmegs < 512) {
                        /* 12% of memory */
                        segpcache_maxwindow = availrmem >> 3;
                } else if (physmegs < 1024) {
                        /* 25% of memory */
                        segpcache_maxwindow = availrmem >> 2;
                } else if (physmegs < 2048) {
                        /* 50% of memory */
                        segpcache_maxwindow = availrmem >> 1;
                } else {
                        /* no limit */
                        segpcache_maxwindow = (pgcnt_t)-1;
                }
        }
        seg_pmaxwindow = segpcache_maxwindow;
        seg_pinit_mem_config();
}

/*
 * called by pageout if memory is low
 */
void
seg_preap(void)
{
        /*
         * if the cache is off or empty, return
         */
        if (seg_plocked_window == 0) {
                return;
        }
        ASSERT(seg_phashsize_win != 0);

        /*
         * If somebody is already purging pcache
         * just return.
```

```
         */
        if (seg_pdisabled) {
                return;
        }

        cv_signal(&seg_pasync_cv);
}

/*
 * run as a backgroud thread and reclaim pagelock
 * pages which have not been used recently
 */
void
seg_pasync_thread(void)
{
        callb_cpr_t cpr_info;

        if (seg_phashsize_win == 0) {
                thread_exit();
                /*NOTREACHED*/
        }

        seg_pasync_thr = curthread;

        CALLB_CPR_INIT(&cpr_info, &seg_pasync_mtx,
            callb_generic_cpr, "seg_pasync");

        if (segpcache_reap_ticks <= 0) {
                segpcache_reap_ticks = segpcache_reap_sec * hz;
        }

        mutex_enter(&seg_pasync_mtx);
        for (;;) {
                CALLB_CPR_SAFE_BEGIN(&cpr_info);
                (void) cv_reltimedwait(&seg_pasync_cv, &seg_pasync_mtx,
                    segpcache_reap_ticks, TR_CLOCK_TICK);
                CALLB_CPR_SAFE_END(&cpr_info, &seg_pasync_mtx);
                if (seg_pdisabled == 0) {
                        seg_ppurge_async(0);
                }
        }
}

static struct kmem_cache *seg_cache;

/*
 * Initialize segment management data structures.
 */
void
seg_init(void)
{
        kstat_t *ksp;

        seg_cache = kmem_cache_create("seg_cache", sizeof (struct seg),
            0, NULL, NULL, NULL, NULL, NULL, 0);

        ksp = kstat_create("unix", 0, "segadvstat", "vm", KSTAT_TYPE_NAMED,
            segadvstat_ndata, KSTAT_FLAG_VIRTUAL);
        if (ksp) {
                ksp->ks_data = (void *)segadvstat_ptr;
                kstat_install(ksp);
        }

        seg_pinit();
}
```

```c
/*
 * Allocate a segment to cover [base, base+size]
 * and attach it to the specified address space.
 */
struct seg *
seg_alloc(struct as *as, caddr_t base, size_t size)
{
        struct seg *new;
        caddr_t segbase;
        size_t segsize;

        segbase = (caddr_t)((uintptr_t)base & (uintptr_t)PAGEMASK);
        segsize = (((uintptr_t)(base + size) + PAGEOFFSET) & PAGEMASK) -
            (uintptr_t)segbase;

        if (!valid_va_range(&segbase, &segsize, segsize, AH_LO))
                return ((struct seg *)NULL);    /* bad virtual addr range */

        if (as != &kas &&
            valid_usr_range(segbase, segsize, 0, as,
            as->a_userlimit) != RANGE_OKAY)
                return ((struct seg *)NULL);    /* bad virtual addr range */

        new = kmem_cache_alloc(seg_cache, KM_SLEEP);
        new->s_ops = NULL;
        new->s_data = NULL;
        new->s_szc = 0;
        new->s_flags = 0;
        mutex_init(&new->s_pmtx, NULL, MUTEX_DEFAULT, NULL);
        new->s_phead.p_lnext = &new->s_phead;
        new->s_phead.p_lprev = &new->s_phead;
        if (seg_attach(as, segbase, segsize, new) < 0) {
                kmem_cache_free(seg_cache, new);
                return ((struct seg *)NULL);
        }
        /* caller must fill in ops, data */
        return (new);
}

/*
 * Attach a segment to the address space.  Used by seg_alloc()
 * and for kernel startup to attach to static segments.
 */
int
seg_attach(struct as *as, caddr_t base, size_t size, struct seg *seg)
{
        seg->s_as = as;
        seg->s_base = base;
        seg->s_size = size;

        /*
         * as_addseg() will add the segment at the appropraite point
         * in the list. It will return -1 if there is overlap with
         * an already existing segment.
         */
        return (as_addseg(as, seg));
}

/*
 * Unmap a segment and free it from its associated address space.
 * This should be called by anybody who’s finished with a whole segment’s
 * mapping.  Just calls SEGOP_UNMAP() on the whole mapping .  It is the
 * responsibility of the segment driver to unlink the the segment
 * from the address space, and to free public and private data structures
 * associated with the segment.  (This is typically done by a call to
 * seg_free()).
```

```c
 */
void
seg_unmap(struct seg *seg)
{
#ifdef DEBUG
        int ret;
#endif /* DEBUG */

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

        /* Shouldn’t have called seg_unmap if mapping isn’t yet established */
        ASSERT(seg->s_data != NULL);

        /* Unmap the whole mapping */
#ifdef DEBUG
        ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
        ASSERT(ret == 0);
#else
        SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
#endif /* DEBUG */
}

/*
 * Free the segment from its associated as. This should only be called
 * if a mapping to the segment has not yet been established (e.g., if
 * an error occurs in the middle of doing an as_map when the segment
 * has already been partially set up) or if it has already been deleted
 * (e.g., from a segment driver unmap routine if the unmap applies to the
 * entire segment). If the mapping is currently set up then seg_unmap() should
 * be called instead.
 */
void
seg_free(struct seg *seg)
{
        register struct as *as = seg->s_as;
        struct seg *tseg = as_removeseg(as, seg);

        ASSERT(tseg == seg);

        /*
         * If the segment private data field is NULL,
         * then segment driver is not attached yet.
         */
        if (seg->s_data != NULL)
                SEGOP_FREE(seg);

        mutex_destroy(&seg->s_pmtx);
        ASSERT(seg->s_phead.p_lnext == &seg->s_phead);
        ASSERT(seg->s_phead.p_lprev == &seg->s_phead);
        kmem_cache_free(seg_cache, seg);
}

/*ARGSUSED*/
static void
seg_p_mem_config_post_add(
        void *arg,
        pgcnt_t delta_pages)
{
        /* Nothing to do. */
}

void
seg_p_enable(void)
{
        mutex_enter(&seg_pcache_mtx);
        ASSERT(seg_pdisabled != 0);
        seg_pdisabled--;
        mutex_exit(&seg_pcache_mtx);
}

/*
 * seg_p_disable - disables seg_pcache, and then attempts to empty the
 * cache.
 * Returns SEGP_SUCCESS if the cache was successfully emptied, or
 * SEGP_FAIL if the cache could not be emptied.
 */
int
seg_p_disable(void)
{
        pgcnt_t old_plocked;
        int stall_count = 0;

        mutex_enter(&seg_pcache_mtx);
        seg_pdisabled++;
        ASSERT(seg_pdisabled != 0);
        mutex_exit(&seg_pcache_mtx);

        /*
         * Attempt to empty the cache. Terminate if seg_plocked does not
         * diminish with SEGP_STALL_THRESHOLD consecutive attempts.
         */
        while (seg_plocked != 0) {
                ASSERT(seg_phashsize_win != 0);
                old_plocked = seg_plocked;
                seg_ppurge_async(1);
                if (seg_plocked == old_plocked) {
                        if (stall_count++ > SEGP_STALL_THRESHOLD) {
                                return (SEGP_FAIL);
                        }
                } else
                        stall_count = 0;
                if (seg_plocked != 0)
                        delay(hz/SEGP_PREDEL_DELAY_FACTOR);
        }
        return (SEGP_SUCCESS);
}

/*
 * Attempt to purge seg_pcache.  May need to return before this has
 * completed to allow other pre_del callbacks to unlock pages. This is
 * ok because:
 *      1) The seg_pdisabled flag has been set so at least we won’t
 *      cache anymore locks and the locks we couldn’t purge
 *      will not be held if they do get released by a subsequent
 *      pre-delete callback.
 *
 *      2) The rest of the memory delete thread processing does not
 *      depend on the changes made in this pre-delete callback. No
 *      panics will result, the worst that will happen is that the
 *      DR code will timeout and cancel the delete.
 */
/*ARGSUSED*/
static int
seg_p_mem_config_pre_del(
        void *arg,
        pgcnt_t delta_pages)
{
        if (seg_phashsize_win == 0) {
                return (0);
        }
        if (seg_p_disable() != SEGP_SUCCESS)
                cmn_err(CE_NOTE,
```

```
                    "!Pre-delete couldn’t purge"" pagelock cache - continuing");
        return (0);
}

/*ARGSUSED*/
static void
seg_p_mem_config_post_del(
        void *arg,
        pgcnt_t delta_pages,
        int cancelled)
{
        if (seg_phashsize_win == 0) {
                return;
        }
        seg_p_enable();
}

static kphysm_setup_vector_t seg_p_mem_config_vec = {
        KPHYSM_SETUP_VECTOR_VERSION,
        seg_p_mem_config_post_add,
        seg_p_mem_config_pre_del,
        seg_p_mem_config_post_del,
};

static void
seg_pinit_mem_config(void)
{
        int ret;

        ret = kphysm_setup_func_register(&seg_p_mem_config_vec, (void *)NULL);
        /*
         * Want to catch this in the debug kernel. At run time, if the
         * callbacks don’t get run all will be OK as the disable just makes
         * it more likely that the pages can be collected.
         */
        ASSERT(ret == 0);
}

/*
 * Verify that segment is not a shared anonymous segment which reserves
 * swap.  zone.max-swap accounting (zone->zone_max_swap) cannot be transfered
 * from one zone to another if any segments are shared.  This is because the
 * last process to exit will credit the swap reservation.  This could lead
 * to the swap being reserved by one zone, and credited to another.
 */
boolean_t
seg_can_change_zones(struct seg *seg)
{
        struct segvn_data *svd;

        if (seg->s_ops == &segspt_shmops)
                return (B_FALSE);

        if (seg->s_ops == &segvn_ops) {
                svd = (struct segvn_data *)seg->s_data;
                if (svd->type == MAP_SHARED &&
                    svd->amp != NULL &&
                    svd->amp->swresv > 0)
                return (B_FALSE);
        }
        return (B_TRUE);
}

/*
 * Return swap reserved by a segment backing a private mapping.
 */
```

```
size_t
seg_swresv(struct seg *seg)
{
        struct segvn_data *svd;
        size_t swap = 0;

        if (seg->s_ops == &segvn_ops) {
                svd = (struct segvn_data *)seg->s_data;
                if (svd->type == MAP_PRIVATE && svd->swresv > 0)
                        swap = svd->swresv;
        }
        return (swap);
}

/*
 * General not supported function for SEGOP_INHERIT
 */
/* ARGSUSED */
int
seg_inherit_notsup(struct seg *seg, caddr_t addr, size_t len, uint_t op)
{
        return (ENOTSUP);
}

/*
 * segop wrappers
 */
int
segop_dup(struct seg *seg, struct seg *new)
{
        return (seg->s_ops->dup(seg, new));
}

int
segop_unmap(struct seg *seg, caddr_t addr, size_t len)
{
        return (seg->s_ops->unmap(seg, addr, len));
}

void
segop_free(struct seg *seg)
{
        seg->s_ops->free(seg);
}

faultcode_t
segop_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
    enum fault_type type, enum seg_rw rw)
{
        return (seg->s_ops->fault(hat, seg, addr, len, type, rw));
}

faultcode_t
segop_faulta(struct seg *seg, caddr_t addr)
{
        return (seg->s_ops->faulta(seg, addr));
}

int
segop_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
{
        return (seg->s_ops->setprot(seg, addr, len, prot));
}

int
segop_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
```

```
{
        return (seg->s_ops->checkprot(seg, addr, len, prot));
}

int
segop_kluster(struct seg *seg, caddr_t addr, ssize_t d)
{
        return (seg->s_ops->kluster(seg, addr, d));
}

size_t
segop_swapout(struct seg *seg)
{
        return (seg->s_ops->swapout(seg));
}

int
segop_sync(struct seg *seg, caddr_t addr, size_t len, int atr, uint_t f)
{
        return (seg->s_ops->sync(seg, addr, len, atr, f));
}

size_t
segop_incore(struct seg *seg, caddr_t addr, size_t len, char *v)
{
        return (seg->s_ops->incore(seg, addr, len, v));
}

int
segop_lockop(struct seg *seg, caddr_t addr, size_t len, int atr, int op,
    ulong_t *b, size_t p)
{
        return (seg->s_ops->lockop(seg, addr, len, atr, op, b, p));
}

int
segop_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *p)
{
        return (seg->s_ops->getprot(seg, addr, len, p));
}

u_offset_t
segop_getoffset(struct seg *seg, caddr_t addr)
{
        return (seg->s_ops->getoffset(seg, addr));
}

int
segop_gettype(struct seg *seg, caddr_t addr)
{
        return (seg->s_ops->gettype(seg, addr));
}

int
segop_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
{
        return (seg->s_ops->getvp(seg, addr, vpp));
}

int
segop_advise(struct seg *seg, caddr_t addr, size_t len, uint_t b)
{
        return (seg->s_ops->advise(seg, addr, len, b));
}

void
```

```
segop_dump(struct seg *seg)
{
        seg->s_ops->dump(seg);
}

int
segop_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***page,
    enum lock_type type, enum seg_rw rw)
{
        return (seg->s_ops->pagelock(seg, addr, len, page, type, rw));
}

int
segop_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
{
        return (seg->s_ops->setpagesize(seg, addr, len, szc));
}

int
segop_getmemid(struct seg *seg, caddr_t addr, memid_t *mp)
{
        return (seg->s_ops->getmemid(seg, addr, mp));
}

struct lgrp_mem_policy_info *
segop_getpolicy(struct seg *seg, caddr_t addr)
{
        if (seg->s_ops->getpolicy == NULL)
                return (NULL);

        return (seg->s_ops->getpolicy(seg, addr));
}

int
segop_capable(struct seg *seg, segcapability_t cap)
{
        return (seg->s_ops->capable(seg, cap));
}

int
segop_inherit(struct seg *seg, caddr_t addr, size_t len, uint_t op)
{
        if (seg->s_ops->inherit == NULL)
                return (ENOTSUP);

        return (seg->s_ops->inherit(seg, addr, len, op));
#endif /* ! codereview */
}
```