```
**********************************************************
   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_

 147 #ifdef _KERNEL

 149 /*
 150  * Generic segment operations
 151  */
 152 extern  void    seg_init(void);
 153 extern  struct  seg *seg_alloc(struct as *as, caddr_t base, size_t size);
 154 extern  int     seg_attach(struct as *as, caddr_t base, size_t size,
 155                         struct seg *seg);
 156 extern  void    seg_unmap(struct seg *seg);
 157 extern  void    seg_free(struct seg *seg);

 159 /*
 160  * functions for pagelock cache support
 161  */
 162 typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
 163     struct page **, enum seg_rw, int);

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

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

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

 181 extern  void    seg_pasync_thread(void);
 182 extern  void    seg_preap(void);
 183 extern  int     seg_p_disable(void);
 184 extern  void    seg_p_enable(void);

 186 extern  segadvstat_t    segadvstat;

 188 /*
 189  * Flags for pagelock cache support.
 190  * Flags argument is passed as uint_t to pcache routines.  upper 16 bits of
 191  * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
 192  * is set.
 193  */
 194 #define SEGP_FORCE_WIRED        0x1     /* skip check against seg_pwindow */
 195 #define SEGP_AMP                0x2     /* anon map's pcache entry */
 196 #define SEGP_PSHIFT             0x4     /* addr pgsz shift for hash function */

 198 /*
 199  * Return values for seg_pinsert and seg_pinsert_check functions.
 200  */
 201 #define SEGP_SUCCESS            0       /* seg_pinsert() succeeded */
 202 #define SEGP_FAIL               1       /* seg_pinsert() failed */

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



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

 213 #define SEGOP_DUP(s, n)             (*(s)->s_ops->dup)((s), (n))
 214 #define SEGOP_UNMAP(s, a, l)        (*(s)->s_ops->unmap)((s), (a), (l))
 215 #define SEGOP_FREE(s)               (*(s)->s_ops->free)((s))
 216 #define SEGOP_FAULT(h, s, a, l, t, rw) \
 217                 (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
 218 #define SEGOP_FAULTA(s, a)          (*(s)->s_ops->faulta)((s), (a))
 219 #define SEGOP_SETPROT(s, a, l, p)   (*(s)->s_ops->setprot)((s), (a), (l), (p))
 220 #define SEGOP_CHECKPROT(s, a, l, p) (*(s)->s_ops->checkprot)((s), (a), (l), (p))
 221 #define SEGOP_KLUSTER(s, a, d)      (*(s)->s_ops->kluster)((s), (a), (d))
 222 #define SEGOP_SWAPOUT(s)            (*(s)->s_ops->swapout)((s))
 223 #define SEGOP_SYNC(s, a, l, atr, f) \
 224                 (*(s)->s_ops->sync)((s), (a), (l), (atr), (f))
 225 #define SEGOP_INCORE(s, a, l, v)    (*(s)->s_ops->incore)((s), (a), (l), (v))
 226 #define SEGOP_LOCKOP(s, a, l, atr, op, b, p) \
 227                 (*(s)->s_ops->lockop)((s), (a), (l), (atr), (op), (b), (p))
 228 #define SEGOP_GETPROT(s, a, l, p)   (*(s)->s_ops->getprot)((s), (a), (l), (p))
 229 #define SEGOP_GETOFFSET(s, a)       (*(s)->s_ops->getoffset)((s), (a))
 230 #define SEGOP_GETTYPE(s, a)         (*(s)->s_ops->gettype)((s), (a))
 231 #define SEGOP_GETVP(s, a, vpp)      (*(s)->s_ops->getvp)((s), (a), (vpp))
 232 #define SEGOP_ADVISE(s, a, l, b)    (*(s)->s_ops->advise)((s), (a), (l), (b))
 233 #define SEGOP_DUMP(s)               (*(s)->s_ops->dump)((s))
 234 #define SEGOP_PAGELOCK(s, a, l, p, t, rw) \
 235                 (*(s)->s_ops->pagelock)((s), (a), (l), (p), (t), (rw))
 236 #define SEGOP_SETPAGESIZE(s, a, l, szc) \
 237                 (*(s)->s_ops->setpagesize)((s), (a), (l), (szc))
 238 #define SEGOP_GETMEMID(s, a, mp)    (*(s)->s_ops->getmemid)((s), (a), (mp))
 239 #define SEGOP_GETPOLICY(s, a)       (*(s)->s_ops->getpolicy)((s), (a))
 240 #define SEGOP_CAPABLE(s, c)         (*(s)->s_ops->capable)((s), (c))
 241 #define SEGOP_INHERIT(s, a, l, b)   (*(s)->s_ops->inherit)((s), (a), (l), (b))

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

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

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

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

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

 228 /* Delay/retry factors for seg_p_mem_config_pre_del */
 229 #define SEGP_PREDEL_DELAY_FACTOR        4
 230 /*
 231  * As a workaround to being unable to purge the pagelock
 232  * cache during a DR delete memory operation, we use
 233  * a stall threshold that is twice the maximum seen
 234  * during testing.  This workaround will be removed
 235  * when a suitable fix is found.
 236  */
 237 #define SEGP_STALL_SECONDS      25
 238 #define SEGP_STALL_THRESHOLD \
 239         (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

 241 #ifdef VMDEBUG
```

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

246 #endif  /* VMDEBUG */

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

251 /* segop wrappers */
252 extern int segop_dup(struct seg *, struct seg *);
253 extern int segop_unmap(struct seg *, caddr_t, size_t);
254 extern void segop_free(struct seg *);
255 extern faultcode_t segop_fault(struct hat *, struct seg *, caddr_t, size_t,
256     enum fault_type, enum seg_rw);
257 extern faultcode_t segop_faulta(struct seg *, caddr_t);
258 extern int segop_setprot(struct seg *, caddr_t, size_t, uint_t);
259 extern int segop_checkprot(struct seg *, caddr_t, size_t, uint_t);
260 extern int segop_kluster(struct seg *, caddr_t, ssize_t);
261 extern size_t segop_swapout(struct seg *);
262 extern int segop_sync(struct seg *, caddr_t, size_t, int, uint_t);
263 extern size_t segop_incore(struct seg *, caddr_t, size_t, char *);
264 extern int segop_lockop(struct seg *, caddr_t, size_t, int, int, ulong_t *,
265     size_t);
266 extern int segop_getprot(struct seg *, caddr_t, size_t, uint_t *);
267 extern u_offset_t segop_getoffset(struct seg *, caddr_t);
268 extern int segop_gettype(struct seg *, caddr_t);
269 extern int segop_getvp(struct seg *, caddr_t, struct vnode **);
270 extern int segop_advise(struct seg *, caddr_t, size_t, uint_t);
271 extern void segop_dump(struct seg *);
272 extern int segop_pagelock(struct seg *, caddr_t, size_t, struct page ***,
273     enum lock_type, enum seg_rw);
274 extern int segop_setpagesize(struct seg *, caddr_t, size_t, uint_t);
275 extern int segop_getmemid(struct seg *, caddr_t, memid_t *);
276 extern struct lgrp_mem_policy_info *segop_getpolicy(struct seg *, caddr_t);
277 extern int segop_capable(struct seg *, segcapability_t);
278 extern int segop_inherit(struct seg *, caddr_t, size_t, uint_t);
279 #endif /* ! codereview */

281 #endif  /* _KERNEL */

283 #ifdef  __cplusplus
284 }
285 #endif

287 #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_maxapurge_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;
        }
```

```
392         ASSERT(hp->p_halink[lix].p_lprev == NULL);

394         /*
395          * If asynchronous thread is not running we can remove the bucket from
396          * not currently active list. The bucket must be on this list since we
397          * already checked that it's not on the other list and the bucket from
398          * which we just deleted the last pcp entry must be still on one of the
399          * active bucket lists.
400          */
401         lix = !lix;
402         ASSERT(hp->p_halink[lix].p_lnext != NULL);
403         ASSERT(hp->p_halink[lix].p_lprev != NULL);

405         if (!seg_pathr_on) {
406                 hp->p_halink[lix].p_lnext->p_lprev = hp->p_halink[lix].p_lprev;
407                 hp->p_halink[lix].p_lprev->p_lnext = hp->p_halink[lix].p_lnext;
408                 hp->p_halink[lix].p_lnext = NULL;
409                 hp->p_halink[lix].p_lprev = NULL;
410         }
411         mutex_exit(&seg_pmem_mtx);
412 }

414 /*
415  * Check if bucket pointed by hp already has a pcp entry that matches request
416  * htag0, addr and len. Set *found to 1 if match is found and to 0 otherwise.
417  * Also delete matching entries that cover smaller address range but start
418  * at the same address as addr argument. Return the list of deleted entries if
419  * any. This is an internal helper function called from seg_pinsert() only
420  * for non wired shadow lists. The caller already holds a per seg/amp list
421  * lock.
422  */
423 static struct seg_pcache *
424 seg_plookup_checkdup(struct seg_phash *hp, void *htag0,
425     caddr_t addr, size_t len, int *found)
426 {
427         struct seg_pcache *pcp;
428         struct seg_pcache *delcallb_list = NULL;

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

432         *found = 0;
433         for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
434             pcp = pcp->p_hnext) {
435                 ASSERT(pcp->p_hashp == hp);
436                 if (pcp->p_htag0 == htag0 && pcp->p_addr == addr) {
437                         ASSERT(!IS_PCP_WIRED(pcp));
438                         if (pcp->p_len < len) {
439                                 pcache_link_t *plinkp;
440                                 if (pcp->p_active) {
441                                         continue;
442                                 }
443                                 plinkp = &pcp->p_plink;
444                                 plinkp->p_lprev->p_lnext = plinkp->p_lnext;
445                                 plinkp->p_lnext->p_lprev = plinkp->p_lprev;
446                                 pcp->p_hprev->p_hnext = pcp->p_hnext;
447                                 pcp->p_hnext->p_hprev = pcp->p_hprev;
448                                 pcp->p_hprev = delcallb_list;
449                                 delcallb_list = pcp;
450                         } else {
451                                 *found = 1;
452                                 break;
453                         }
454                 }
455         }
456         return (delcallb_list);
457 }
```



```
459 /*
460  * lookup an address range in pagelock cache. Return shadow list and bump up
461  * active count. If amp is not NULL use amp as a lookup tag otherwise use seg
462  * as a lookup tag.
463  */
464 struct page **
465 seg_plookup(struct seg *seg, struct anon_map *amp, caddr_t addr, size_t len,
466     enum seg_rw rw, uint_t flags)
467 {
468         struct seg_pcache *pcp;
469         struct seg_phash *hp;
470         void *htag0;

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

475         /*
476          * Skip pagelock cache, while DR is in progress or
477          * seg_pcache is off.
478          */
479         if (seg_pdisabled) {
480                 return (NULL);
481         }
482         ASSERT(seg_phashsize_win != 0);

484         htag0 = (amp == NULL ? (void *)seg : (void *)amp);
485         hp = P_HASHBP(seg, htag0, addr, flags);
486         mutex_enter(&hp->p_hmutex);
487         for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
488             pcp = pcp->p_hnext) {
489                 ASSERT(pcp->p_hashp == hp);
490                 if (P_MATCH(pcp, htag0, addr, len)) {
491                         ASSERT(IS_PFLAGS_WIRED(flags) == IS_PCP_WIRED(pcp));
492                         /*
493                          * If this request wants to write pages
494                          * but write permissions starting from
495                          * addr don't cover the entire length len
496                          * return lookup failure back to the caller.
497                          * It will check protections and fail this
498                          * pagelock operation with EACCESS error.
499                          */
500                         if (rw == S_WRITE && pcp->p_wlen < len) {
501                                 break;
502                         }
503                         if (pcp->p_active == UINT_MAX) {
504                                 break;
505                         }
506                         pcp->p_active++;
507                         if (rw == S_WRITE && !pcp->p_write) {
508                                 pcp->p_write = 1;
509                         }
510                         mutex_exit(&hp->p_hmutex);
511                         return (pcp->p_pp);
512                 }
513         }
514         mutex_exit(&hp->p_hmutex);
515         return (NULL);
516 }

518 /*
519  * mark address range inactive. If the cache is off or the address range is
520  * not in the cache or another shadow list that covers bigger range is found
521  * we call the segment driver to reclaim the pages. Otherwise just decrement
522  * active count and set ref bit.  If amp is not NULL use amp as a lookup tag
523  * 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;
                        }
```

```
656                 } else if (!IS_PFLAGS_WIRED(flags) &&
657                     P_MATCH(pcp, htag0, addr, len)) {
658                         /*
659                          * This is a duplicate pcp entry.  This situation may
660                          * happen if a bigger shadow list that covers our
661                          * range was added while our entry was still active.
662                          * Now we can free our pcp entry if it becomes
663                          * inactive.
664                          */
665                         if (!pcp->p_active) {
666                                 /*
667                                  * Mark this entry as referenced just in case
668                                  * we'll free our own pcp entry soon.
669                                  */
670                                 pcp->p_lbolt = ddi_get_lbolt();
671                                 pcp->p_ref = 1;
672                         }
673                         if (pmtx != NULL) {
674                                 /*
675                                  * we are already holding pmtx and found a
676                                  * duplicate.  Don't keep our own pcp entry.
677                                  */
678                                 keep = 0;
679                                 continue;
680                         }
681                         /*
682                          * We have to use mutex_tryenter to attempt to lock
683                          * seg/amp list lock since we already hold hash lock
684                          * and seg/amp list lock is above hash lock in lock
685                          * order.  If mutex_tryenter fails drop hash lock and
686                          * retake both locks in correct order and research
687                          * this hash chain.
688                          */
689                         ASSERT(keep == 0);
690                         if (amp == NULL) {
691                                 pheadp = &seg->s_phead;
692                                 pmtx = &seg->s_pmtx;
693                         } else {
694                                 pheadp = &amp->a_phead;
695                                 pmtx = &amp->a_pmtx;
696                         }
697                         if (!mutex_tryenter(pmtx)) {
698                                 mutex_exit(&hp->p_hmutex);
699                                 mutex_enter(pmtx);
700                                 mutex_enter(&hp->p_hmutex);
701                                 /*
702                                  * If we don't find bigger shadow list on
703                                  * second search (it may happen since we
704                                  * dropped bucket lock) keep the entry that
705                                  * matches our own shadow list.
706                                  */
707                                 keep = 1;
708                                 goto again;
709                         }
710                 }
711         }
712         mutex_exit(&hp->p_hmutex);
713         if (pmtx != NULL) {
714                 mutex_exit(pmtx);
715         }
716 out:
717         (*callback)(htag0, addr, len, pp, rw, 0);
718         if (npages) {
719                 mutex_enter(&seg_pmem_mtx);
720                 ASSERT(seg_plocked >= npages);
721                 seg_plocked -= npages;
```



```
722                 if (!IS_PFLAGS_WIRED(flags)) {
723                         ASSERT(seg_plocked_window >= npages);
724                         seg_plocked_window -= npages;
725                 }
726                 mutex_exit(&seg_pmem_mtx);
727         }

729 }

731 #ifdef DEBUG
732 static uint32_t p_insert_chk_mtbf = 0;
733 #endif

735 /*
736  * The seg_pinsert_check() is used by segment drivers to predict whether
737  * a call to seg_pinsert will fail and thereby avoid wasteful pre-processing.
738  */
739 /*ARGSUSED*/
740 int
741 seg_pinsert_check(struct seg *seg, struct anon_map *amp, caddr_t addr,
742     size_t len, uint_t flags)
743 {
744         ASSERT(seg != NULL);

746 #ifdef DEBUG
747         if (p_insert_chk_mtbf && !(gethrtime() % p_insert_chk_mtbf)) {
748                 return (SEGP_FAIL);
749         }
750 #endif

752         if (seg_pdisabled) {
753                 return (SEGP_FAIL);
754         }
755         ASSERT(seg_phashsize_win != 0);

757         if (IS_PFLAGS_WIRED(flags)) {
758                 return (SEGP_SUCCESS);
759         }

761         if (seg_plocked_window + btop(len) > seg_pmaxwindow) {
762                 return (SEGP_FAIL);
763         }

765         if (freemem < desfree) {
766                 return (SEGP_FAIL);
767         }

769         return (SEGP_SUCCESS);
770 }

772 #ifdef DEBUG
773 static uint32_t p_insert_mtbf = 0;
774 #endif

776 /*
777  * Insert address range with shadow list into pagelock cache if there's no
778  * shadow list already cached for this address range. If the cache is off or
779  * caching is temporarily disabled or the allowed 'window' is exceeded return
780  * SEGP_FAIL. Otherwise return SEGP_SUCCESS.
781  *
782  * For non wired shadow lists (segvn case) include address in the hashing
783  * function to avoid linking all the entries from the same segment or amp on
784  * the same bucket.  amp is used instead of seg if amp is not NULL. Non wired
785  * pcache entries are also linked on a per segment/amp list so that all
786  * entries can be found quickly during seg/amp purge without walking the
787  * 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);
```

```
920         }
921         if (npages) {
922                 ASSERT(!IS_PFLAGS_WIRED(flags));
923                 mutex_enter(&seg_pmem_mtx);
924                 ASSERT(seg_plocked >= npages);
925                 ASSERT(seg_plocked_window >= npages);
926                 seg_plocked -= npages;
927                 seg_plocked_window -= npages;
928                 mutex_exit(&seg_pmem_mtx);
929         }

931         return (SEGP_SUCCESS);
932 }

934 /*
935  * purge entries from the pagelock cache if not active
936  * and not recently used.
937  */
938 static void
939 seg_ppurge_async(int force)
940 {
941         struct seg_pcache *delcallb_list = NULL;
942         struct seg_pcache *pcp;
943         struct seg_phash *hp;
944         pgcnt_t npages = 0;
945         pgcnt_t npages_window = 0;
946         pgcnt_t npgs_to_purge;
947         pgcnt_t npgs_purged = 0;
948         int hlinks = 0;
949         int hlix;
950         pcache_link_t *hlinkp;
951         pcache_link_t *hlnextp = NULL;
952         int lowmem;
953         int trim;

955         ASSERT(seg_phashsize_win != 0);

957         /*
958          * if the cache is off or empty, return
959          */
960         if (seg_plocked == 0 || (!force && seg_plocked_window == 0)) {
961                 return;
962         }

964         if (!force) {
965                 lowmem = 0;
966                 trim = 0;
967                 if (freemem < lotsfree + needfree) {
968                         spgcnt_t fmem = MAX((spgcnt_t)(freemem - needfree), 0);
969                         if (fmem <= 5 * (desfree >> 2)) {
970                                 lowmem = 1;
971                         } else if (fmem <= 7 * (lotsfree >> 3)) {
972                                 if (seg_plocked_window >=
973                                     (availrmem_initial >> 1)) {
974                                         lowmem = 1;
975                                 }
976                         } else if (fmem < lotsfree) {
977                                 if (seg_plocked_window >=
978                                     3 * (availrmem_initial >> 2)) {
979                                         lowmem = 1;
980                                 }
981                         }
982                 }
983                 if (seg_plocked_window >= 7 * (seg_pmaxwindow >> 3)) {
984                         trim = 1;
985                 }
```



```
986                 if (!lowmem && !trim) {
987                         return;
988                 }
989                 npgs_to_purge = seg_plocked_window >>
990                     seg_pshrink_shift;
991                 if (lowmem) {
992                         npgs_to_purge = MIN(npgs_to_purge,
993                             MAX(seg_pmaxapurge_npages, desfree));
994                 } else {
995                         npgs_to_purge = MIN(npgs_to_purge,
996                             seg_pmaxapurge_npages);
997                 }
998                 if (npgs_to_purge == 0) {
999                         return;
1000                }
1001        } else {
1002                struct seg_phash_wired *hpw;

1004                ASSERT(seg_phashsize_wired != 0);

1006                for (hpw = seg_phashtab_wired;
1007                    hpw < &seg_phashtab_wired[seg_phashsize_wired]; hpw++) {

1009                        if (hpw->p_hnext == (struct seg_pcache *)hpw) {
1010                                continue;
1011                        }

1013                        mutex_enter(&hpw->p_hmutex);

1015                        for (pcp = hpw->p_hnext;
1016                            pcp != (struct seg_pcache *)hpw;
1017                            pcp = pcp->p_hnext) {

1019                                ASSERT(IS_PCP_WIRED(pcp));
1020                                ASSERT(pcp->p_hashp ==
1021                                    (struct seg_phash *)hpw);

1023                                if (pcp->p_active) {
1024                                        continue;
1025                                }
1026                                pcp->p_hprev->p_hnext = pcp->p_hnext;
1027                                pcp->p_hnext->p_hprev = pcp->p_hprev;
1028                                pcp->p_hprev = delcallb_list;
1029                                delcallb_list = pcp;
1030                        }
1031                        mutex_exit(&hpw->p_hmutex);
1032                }
1033        }

1035        mutex_enter(&seg_pmem_mtx);
1036        if (seg_pathr_on) {
1037                mutex_exit(&seg_pmem_mtx);
1038                goto runcb;
1039        }
1040        seg_pathr_on = 1;
1041        mutex_exit(&seg_pmem_mtx);
1042        ASSERT(seg_pahcur <= 1);
1043        hlix = !seg_pahcur;

1045 again:
1046        for (hlinkp = seg_pahhead[hlix].p_lnext; hlinkp != &seg_pahhead[hlix];
1047            hlinkp = hlnextp) {

1049                hlnextp = hlinkp->p_lnext;
1050                ASSERT(hlnextp != NULL);
```

```
1052                hp = hlink2phash(hlinkp, hlix);
1053                if (hp->p_hnext == (struct seg_pcache *)hp) {
1054                        seg_pathr_empty_ahb++;
1055                        continue;
1056                }
1057                seg_pathr_full_ahb++;
1058                mutex_enter(&hp->p_hmutex);

1060                for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
1061                    pcp = pcp->p_hnext) {
1062                        pcache_link_t *pheadp;
1063                        pcache_link_t *plinkp;
1064                        void *htag0;
1065                        kmutex_t *pmtx;

1067                        ASSERT(!IS_PCP_WIRED(pcp));
1068                        ASSERT(pcp->p_hashp == hp);

1070                        if (pcp->p_active) {
1071                                continue;
1072                        }
1073                        if (!force && pcp->p_ref &&
1074                            PCP_AGE(pcp) < seg_pmax_pcpage) {
1075                                pcp->p_ref = 0;
1076                                continue;
1077                        }
1078                        plinkp = &pcp->p_plink;
1079                        htag0 = pcp->p_htag0;
1080                        if (pcp->p_flags & SEGP_AMP) {
1081                                pheadp = &((amp_t *)htag0)->a_phead;
1082                                pmtx = &((amp_t *)htag0)->a_pmtx;
1083                        } else {
1084                                pheadp = &((seg_t *)htag0)->s_phead;
1085                                pmtx = &((seg_t *)htag0)->s_pmtx;
1086                        }
1087                        if (!mutex_tryenter(pmtx)) {
1088                                continue;
1089                        }
1090                        ASSERT(pheadp->p_lnext != pheadp);
1091                        ASSERT(pheadp->p_lprev != pheadp);
1092                        plinkp->p_lprev->p_lnext =
1093                            plinkp->p_lnext;
1094                        plinkp->p_lnext->p_lprev =
1095                            plinkp->p_lprev;
1096                        pcp->p_hprev->p_hnext = pcp->p_hnext;
1097                        pcp->p_hnext->p_hprev = pcp->p_hprev;
1098                        mutex_exit(pmtx);
1099                        pcp->p_hprev = delcallb_list;
1100                        delcallb_list = pcp;
1101                        npgs_purged += btop(pcp->p_len);
1102                }
1103                if (hp->p_hnext == (struct seg_pcache *)hp) {
1104                        seg_premove_abuck(hp, 1);
1105                }
1106                mutex_exit(&hp->p_hmutex);
1107                if (npgs_purged >= seg_plocked_window) {
1108                        break;
1109                }
1110                if (!force) {
1111                        if (npgs_purged >= npgs_to_purge) {
1112                                break;
1113                        }
1114                        if (!trim && !(seg_pathr_full_ahb & 15)) {
1115                                ASSERT(lowmem);
1116                                if (freemem >= lotsfree + needfree) {
1117                                        break;
```



```
1118                                }
1119                        }
1120                }
1121        }

1123        if (hlinkp == &seg_pahhead[hlix]) {
1124                /*
1125                 * We processed the entire hlix active bucket list
1126                 * but didn't find enough pages to reclaim.
1127                 * Switch the lists and walk the other list
1128                 * if we haven't done it yet.
1129                 */
1130                mutex_enter(&seg_pmem_mtx);
1131                ASSERT(seg_pathr_on);
1132                ASSERT(seg_pahcur == !hlix);
1133                seg_pahcur = hlix;
1134                mutex_exit(&seg_pmem_mtx);
1135                if (++hlinks < 2) {
1136                        hlix = !hlix;
1137                        goto again;
1138                }
1139        } else if ((hlinkp = hlnextp) != &seg_pahhead[hlix] &&
1140            seg_pahhead[hlix].p_lnext != hlinkp) {
1141                ASSERT(hlinkp != NULL);
1142                ASSERT(hlinkp->p_lprev != &seg_pahhead[hlix]);
1143                ASSERT(seg_pahhead[hlix].p_lnext != &seg_pahhead[hlix]);
1144                ASSERT(seg_pahhead[hlix].p_lprev != &seg_pahhead[hlix]);

1146                /*
1147                 * Reinsert the header to point to hlinkp
1148                 * so that we start from hlinkp bucket next time around.
1149                 */
1150                seg_pahhead[hlix].p_lnext->p_lprev = seg_pahhead[hlix].p_lprev;
1151                seg_pahhead[hlix].p_lprev->p_lnext = seg_pahhead[hlix].p_lnext;
1152                seg_pahhead[hlix].p_lnext = hlinkp;
1153                seg_pahhead[hlix].p_lprev = hlinkp->p_lprev;
1154                hlinkp->p_lprev->p_lnext = &seg_pahhead[hlix];
1155                hlinkp->p_lprev = &seg_pahhead[hlix];
1156        }

1158        mutex_enter(&seg_pmem_mtx);
1159        ASSERT(seg_pathr_on);
1160        seg_pathr_on = 0;
1161        mutex_exit(&seg_pmem_mtx);

1163 runcb:
1164        /*
1165         * Run the delayed callback list. segments/amps can't go away until
1166         * callback is executed since they must have non 0 softlockcnt. That's
1167         * why we don't need to hold as/seg/amp locks to execute the callback.
1168         */
1169        while (delcallb_list != NULL) {
1170                pcp = delcallb_list;
1171                delcallb_list = pcp->p_hprev;
1172                ASSERT(!pcp->p_active);
1173                (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
1174                    pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 1);
1175                npages += btop(pcp->p_len);
1176                if (!IS_PCP_WIRED(pcp)) {
1177                        npages_window += btop(pcp->p_len);
1178                }
1179                kmem_cache_free(seg_pkmcache, pcp);
1180        }
1181        if (npages) {
1182                mutex_enter(&seg_pmem_mtx);
1183                ASSERT(seg_plocked >= npages);
```

```
1184                 ASSERT(seg_plocked_window >= npages_window);
1185                 seg_plocked -= npages;
1186                 seg_plocked_window -= npages_window;
1187                 mutex_exit(&seg_pmem_mtx);
1188         }
1189 }

1191 /*
1192  * Remove cached pages for segment(s) entries from hashtable.  The segments
1193  * are identified by pp array. This is useful for multiple seg's cached on
1194  * behalf of dummy segment (ISM/DISM) with common pp array.
1195  */
1196 void
1197 seg_ppurge_wiredpp(struct page **pp)
1198 {
1199         struct seg_pcache *pcp;
1200         struct seg_phash_wired *hp;
1201         pgcnt_t npages = 0;
1202         struct  seg_pcache *delcallb_list = NULL;

1204         /*
1205          * if the cache is empty, return
1206          */
1207         if (seg_plocked == 0) {
1208                 return;
1209         }
1210         ASSERT(seg_phashsize_wired != 0);

1212         for (hp = seg_phashtab_wired;
1213             hp < &seg_phashtab_wired[seg_phashsize_wired]; hp++) {
1214                 if (hp->p_hnext == (struct seg_pcache *)hp) {
1215                         continue;
1216                 }
1217                 mutex_enter(&hp->p_hmutex);
1218                 pcp = hp->p_hnext;
1219                 while (pcp != (struct seg_pcache *)hp) {
1220                         ASSERT(pcp->p_hashp == (struct seg_phash *)hp);
1221                         ASSERT(IS_PCP_WIRED(pcp));
1222                         /*
1223                          * purge entries which are not active
1224                          */
1225                         if (!pcp->p_active && pcp->p_pp == pp) {
1226                                 ASSERT(pcp->p_htag0 != NULL);
1227                                 pcp->p_hprev->p_hnext = pcp->p_hnext;
1228                                 pcp->p_hnext->p_hprev = pcp->p_hprev;
1229                                 pcp->p_hprev = delcallb_list;
1230                                 delcallb_list = pcp;
1231                         }
1232                         pcp = pcp->p_hnext;
1233                 }
1234                 mutex_exit(&hp->p_hmutex);
1235                 /*
1236                  * segments can't go away until callback is executed since
1237                  * they must have non 0 softlockcnt. That's why we don't
1238                  * need to hold as/seg locks to execute the callback.
1239                  */
1240                 while (delcallb_list != NULL) {
1241                         int done;
1242                         pcp = delcallb_list;
1243                         delcallb_list = pcp->p_hprev;
1244                         ASSERT(!pcp->p_active);
1245                         done = (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
1246                             pcp->p_len, pcp->p_pp,
1247                             pcp->p_write ? S_WRITE : S_READ, 1);
1248                         npages += btop(pcp->p_len);
1249                         ASSERT(IS_PCP_WIRED(pcp));
```



```
1250                         kmem_cache_free(seg_pkmcache, pcp);
1251                         if (done) {
1252                                 ASSERT(delcallb_list == NULL);
1253                                 goto out;
1254                         }
1255                 }
1256         }

1258 out:
1259         mutex_enter(&seg_pmem_mtx);
1260         ASSERT(seg_plocked >= npages);
1261         seg_plocked -= npages;
1262         mutex_exit(&seg_pmem_mtx);
1263 }

1265 /*
1266  * purge all entries for a given segment. Since we
1267  * callback into the segment driver directly for page
1268  * reclaim the caller needs to hold the right locks.
1269  */
1270 void
1271 seg_ppurge(struct seg *seg, struct anon_map *amp, uint_t flags)
1272 {
1273         struct seg_pcache *delcallb_list = NULL;
1274         struct seg_pcache *pcp;
1275         struct seg_phash *hp;
1276         pgcnt_t npages = 0;
1277         void *htag0;

1279         if (seg_plocked == 0) {
1280                 return;
1281         }
1282         ASSERT(seg_phashsize_win != 0);

1284         /*
1285          * If amp is not NULL use amp as a lookup tag otherwise use seg
1286          * as a lookup tag.
1287          */
1288         htag0 = (amp == NULL ? (void *)seg : (void *)amp);
1289         ASSERT(htag0 != NULL);
1290         if (IS_PFLAGS_WIRED(flags)) {
1291                 hp = P_HASHBP(seg, htag0, 0, flags);
1292                 mutex_enter(&hp->p_hmutex);
1293                 pcp = hp->p_hnext;
1294                 while (pcp != (struct seg_pcache *)hp) {
1295                         ASSERT(pcp->p_hashp == hp);
1296                         ASSERT(IS_PCP_WIRED(pcp));
1297                         if (pcp->p_htag0 == htag0) {
1298                                 if (pcp->p_active) {
1299                                         break;
1300                                 }
1301                                 pcp->p_hprev->p_hnext = pcp->p_hnext;
1302                                 pcp->p_hnext->p_hprev = pcp->p_hprev;
1303                                 pcp->p_hprev = delcallb_list;
1304                                 delcallb_list = pcp;
1305                         }
1306                         pcp = pcp->p_hnext;
1307                 }
1308                 mutex_exit(&hp->p_hmutex);
1309         } else {
1310                 pcache_link_t *plinkp;
1311                 pcache_link_t *pheadp;
1312                 kmutex_t *pmtx;

1314                 if (amp == NULL) {
1315                         ASSERT(seg != NULL);
```

```
1316                        pheadp = &seg->s_phead;
1317                        pmtx = &seg->s_pmtx;
1318                } else {
1319                        pheadp = &amp->a_phead;
1320                        pmtx = &amp->a_pmtx;
1321                }
1322                mutex_enter(pmtx);
1323                while ((plinkp = pheadp->p_lnext) != pheadp) {
1324                        pcp = plink2pcache(plinkp);
1325                        ASSERT(!IS_PCP_WIRED(pcp));
1326                        ASSERT(pcp->p_htag0 == htag0);
1327                        hp = pcp->p_hashp;
1328                        mutex_enter(&hp->p_hmutex);
1329                        if (pcp->p_active) {
1330                                mutex_exit(&hp->p_hmutex);
1331                                break;
1332                        }
1333                        ASSERT(plinkp->p_lprev == pheadp);
1334                        pheadp->p_lnext = plinkp->p_lnext;
1335                        plinkp->p_lnext->p_lprev = pheadp;
1336                        pcp->p_hprev->p_hnext = pcp->p_hnext;
1337                        pcp->p_hnext->p_hprev = pcp->p_hprev;
1338                        pcp->p_hprev = delcallb_list;
1339                        delcallb_list = pcp;
1340                        if (hp->p_hnext == (struct seg_pcache *)hp) {
1341                                seg_premove_abuck(hp, 0);
1342                        }
1343                        mutex_exit(&hp->p_hmutex);
1344                }
1345                mutex_exit(pmtx);
1346        }
1347        while (delcallb_list != NULL) {
1348                pcp = delcallb_list;
1349                delcallb_list = pcp->p_hprev;
1350                ASSERT(!pcp->p_active);
1351                (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr, pcp->p_len,
1352                    pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 0);
1353                npages += btop(pcp->p_len);
1354                kmem_cache_free(seg_pkmcache, pcp);
1355        }
1356        mutex_enter(&seg_pmem_mtx);
1357        ASSERT(seg_plocked >= npages);
1358        seg_plocked -= npages;
1359        if (!IS_PFLAGS_WIRED(flags)) {
1360                ASSERT(seg_plocked_window >= npages);
1361                seg_plocked_window -= npages;
1362        }
1363        mutex_exit(&seg_pmem_mtx);
1364 }

1366 static void seg_pinit_mem_config(void);

1368 /*
1369  * setup the pagelock cache
1370  */
1371 static void
1372 seg_pinit(void)
1373 {
1374        struct seg_phash *hp;
1375        ulong_t i;
1376        pgcnt_t physmegs;

1378        seg_plocked = 0;
1379        seg_plocked_window = 0;

1381        if (segpcache_enabled == 0) {
```



```
1382                seg_phashsize_win = 0;
1383                seg_phashsize_wired = 0;
1384                seg_pdisabled = 1;
1385                return;
1386        }

1388        seg_pdisabled = 0;
1389        seg_pkmcache = kmem_cache_create("seg_pcache",
1390            sizeof (struct seg_pcache), 0, NULL, NULL, NULL, NULL, NULL, 0);
1391        if (segpcache_pcp_maxage_ticks <= 0) {
1392                segpcache_pcp_maxage_ticks = segpcache_pcp_maxage_sec * hz;
1393        }
1394        seg_pmax_pcpage = segpcache_pcp_maxage_ticks;
1395        seg_pathr_empty_ahb = 0;
1396        seg_pathr_full_ahb = 0;
1397        seg_pshrink_shift = segpcache_shrink_shift;
1398        seg_pmaxapurge_npages = btop(segpcache_maxapurge_bytes);

1400        mutex_init(&seg_pcache_mtx, NULL, MUTEX_DEFAULT, NULL);
1401        mutex_init(&seg_pmem_mtx, NULL, MUTEX_DEFAULT, NULL);
1402        mutex_init(&seg_pasync_mtx, NULL, MUTEX_DEFAULT, NULL);
1403        cv_init(&seg_pasync_cv, NULL, CV_DEFAULT, NULL);

1405        physmegs = physmem >> (20 - PAGESHIFT);

1407        /*
1408         * If segpcache_hashsize_win was not set in /etc/system or it has
1409         * absurd value set it to a default.
1410         */
1411        if (segpcache_hashsize_win == 0 || segpcache_hashsize_win > physmem) {
1412                /*
1413                 * Create one bucket per 32K (or at least per 8 pages) of
1414                 * available memory.
1415                 */
1416                pgcnt_t pages_per_bucket = MAX(btop(32 * 1024), 8);
1417                segpcache_hashsize_win = MAX(1024, physmem / pages_per_bucket);
1418        }
1419        if (!ISP2(segpcache_hashsize_win)) {
1420                ulong_t rndfac = ~(1UL <<
1421                    (highbit(segpcache_hashsize_win) - 1));
1422                rndfac &= segpcache_hashsize_win;
1423                segpcache_hashsize_win += rndfac;
1424                segpcache_hashsize_win = 1 <<
1425                    (highbit(segpcache_hashsize_win) - 1);
1426        }
1427        seg_phashsize_win = segpcache_hashsize_win;
1428        seg_phashtab_win = kmem_zalloc(
1429            seg_phashsize_win * sizeof (struct seg_phash),
1430            KM_SLEEP);
1431        for (i = 0; i < seg_phashsize_win; i++) {
1432                hp = &seg_phashtab_win[i];
1433                hp->p_hnext = (struct seg_pcache *)hp;
1434                hp->p_hprev = (struct seg_pcache *)hp;
1435                mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1436        }

1438        seg_pahcur = 0;
1439        seg_pathr_on = 0;
1440        seg_pahhead[0].p_lnext = &seg_pahhead[0];
1441        seg_pahhead[0].p_lprev = &seg_pahhead[0];
1442        seg_pahhead[1].p_lnext = &seg_pahhead[1];
1443        seg_pahhead[1].p_lprev = &seg_pahhead[1];

1445        /*
1446         * If segpcache_hashsize_wired was not set in /etc/system or it has
1447         * absurd value set it to a default.
```

```
1448          */
1449         if (segpcache_hashsize_wired == 0 ||
1450             segpcache_hashsize_wired > physmem / 4) {
1451                 /*
1452                  * Choose segpcache_hashsize_wired based on physmem.
1453                  * Create a bucket per 128K bytes upto 256K buckets.
1454                  */
1455                 if (physmegs < 20 * 1024) {
1456                         segpcache_hashsize_wired = MAX(1024, physmegs << 3);
1457                 } else {
1458                         segpcache_hashsize_wired = 256 * 1024;
1459                 }
1460         }
1461         if (!ISP2(segpcache_hashsize_wired)) {
1462                 segpcache_hashsize_wired = 1 <<
1463                     highbit(segpcache_hashsize_wired);
1464         }
1465         seg_phashsize_wired = segpcache_hashsize_wired;
1466         seg_phashtab_wired = kmem_zalloc(
1467             seg_phashsize_wired * sizeof (struct seg_phash_wired), KM_SLEEP);
1468         for (i = 0; i < seg_phashsize_wired; i++) {
1469                 hp = (struct seg_phash *)&seg_phashtab_wired[i];
1470                 hp->p_hnext = (struct seg_pcache *)hp;
1471                 hp->p_hprev = (struct seg_pcache *)hp;
1472                 mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1473         }

1475         if (segpcache_maxwindow == 0) {
1476                 if (physmegs < 64) {
1477                         /* 3% of memory */
1478                         segpcache_maxwindow = availrmem >> 5;
1479                 } else if (physmegs < 512) {
1480                         /* 12% of memory */
1481                         segpcache_maxwindow = availrmem >> 3;
1482                 } else if (physmegs < 1024) {
1483                         /* 25% of memory */
1484                         segpcache_maxwindow = availrmem >> 2;
1485                 } else if (physmegs < 2048) {
1486                         /* 50% of memory */
1487                         segpcache_maxwindow = availrmem >> 1;
1488                 } else {
1489                         /* no limit */
1490                         segpcache_maxwindow = (pgcnt_t)-1;
1491                 }
1492         }
1493         seg_pmaxwindow = segpcache_maxwindow;
1494         seg_pinit_mem_config();
1495 }

1497 /*
1498  * called by pageout if memory is low
1499  */
1500 void
1501 seg_preap(void)
1502 {
1503         /*
1504          * if the cache is off or empty, return
1505          */
1506         if (seg_plocked_window == 0) {
1507                 return;
1508         }
1509         ASSERT(seg_phashsize_win != 0);

1511         /*
1512          * If somebody is already purging pcache
1513          * just return.
```



```
1514          */
1515         if (seg_pdisabled) {
1516                 return;
1517         }

1519         cv_signal(&seg_pasync_cv);
1520 }

1522 /*
1523  * run as a backgroud thread and reclaim pagelock
1524  * pages which have not been used recently
1525  */
1526 void
1527 seg_pasync_thread(void)
1528 {
1529         callb_cpr_t cpr_info;

1531         if (seg_phashsize_win == 0) {
1532                 thread_exit();
1533                 /*NOTREACHED*/
1534         }

1536         seg_pasync_thr = curthread;

1538         CALLB_CPR_INIT(&cpr_info, &seg_pasync_mtx,
1539             callb_generic_cpr, "seg_pasync");

1541         if (segpcache_reap_ticks <= 0) {
1542                 segpcache_reap_ticks = segpcache_reap_sec * hz;
1543         }

1545         mutex_enter(&seg_pasync_mtx);
1546         for (;;) {
1547                 CALLB_CPR_SAFE_BEGIN(&cpr_info);
1548                 (void) cv_reltimedwait(&seg_pasync_cv, &seg_pasync_mtx,
1549                     segpcache_reap_ticks, TR_CLOCK_TICK);
1550                 CALLB_CPR_SAFE_END(&cpr_info, &seg_pasync_mtx);
1551                 if (seg_pdisabled == 0) {
1552                         seg_ppurge_async(0);
1553                 }
1554         }
1555 }

1557 static struct kmem_cache *seg_cache;

1559 /*
1560  * Initialize segment management data structures.
1561  */
1562 void
1563 seg_init(void)
1564 {
1565         kstat_t *ksp;

1567         seg_cache = kmem_cache_create("seg_cache", sizeof (struct seg),
1568             0, NULL, NULL, NULL, NULL, NULL, 0);

1570         ksp = kstat_create("unix", 0, "segadvstat", "vm", KSTAT_TYPE_NAMED,
1571             segadvstat_ndata, KSTAT_FLAG_VIRTUAL);
1572         if (ksp) {
1573                 ksp->ks_data = (void *)segadvstat_ptr;
1574                 kstat_install(ksp);
1575         }

1577         seg_pinit();
1578 }
```

```
1580 /*
1581  * Allocate a segment to cover [base, base+size]
1582  * and attach it to the specified address space.
1583  */
1584 struct seg *
1585 seg_alloc(struct as *as, caddr_t base, size_t size)
1586 {
1587         struct seg *new;
1588         caddr_t segbase;
1589         size_t segsize;

1591         segbase = (caddr_t)((uintptr_t)base & (uintptr_t)PAGEMASK);
1592         segsize = (((uintptr_t)(base + size) + PAGEOFFSET) & PAGEMASK) -
1593             (uintptr_t)segbase;

1595         if (!valid_va_range(&segbase, &segsize, segsize, AH_LO))
1596                 return ((struct seg *)NULL);    /* bad virtual addr range */

1598         if (as != &kas &&
1599             valid_usr_range(segbase, segsize, 0, as,
1600             as->a_userlimit) != RANGE_OKAY)
1601                 return ((struct seg *)NULL);    /* bad virtual addr range */

1603         new = kmem_cache_alloc(seg_cache, KM_SLEEP);
1604         new->s_ops = NULL;
1605         new->s_data = NULL;
1606         new->s_szc = 0;
1607         new->s_flags = 0;
1608         mutex_init(&new->s_pmtx, NULL, MUTEX_DEFAULT, NULL);
1609         new->s_phead.p_lnext = &new->s_phead;
1610         new->s_phead.p_lprev = &new->s_phead;
1611         if (seg_attach(as, segbase, segsize, new) < 0) {
1612                 kmem_cache_free(seg_cache, new);
1613                 return ((struct seg *)NULL);
1614         }
1615         /* caller must fill in ops, data */
1616         return (new);
1617 }

1619 /*
1620  * Attach a segment to the address space.  Used by seg_alloc()
1621  * and for kernel startup to attach to static segments.
1622  */
1623 int
1624 seg_attach(struct as *as, caddr_t base, size_t size, struct seg *seg)
1625 {
1626         seg->s_as = as;
1627         seg->s_base = base;
1628         seg->s_size = size;

1630         /*
1631          * as_addseg() will add the segment at the appropraite point
1632          * in the list. It will return -1 if there is overlap with
1633          * an already existing segment.
1634          */
1635         return (as_addseg(as, seg));
1636 }

1638 /*
1639  * Unmap a segment and free it from its associated address space.
1640  * This should be called by anybody who's finished with a whole segment's
1641  * mapping.  Just calls SEGOP_UNMAP() on the whole mapping .  It is the
1642  * responsibility of the segment driver to unlink the the segment
1643  * from the address space, and to free public and private data structures
1644  * associated with the segment.  (This is typically done by a call to
1645  * seg_free()).
```



```
1646  */
1647 void
1648 seg_unmap(struct seg *seg)
1649 {
1650 #ifdef DEBUG
1651         int ret;
1652 #endif /* DEBUG */

1654         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1656         /* Shouldn't have called seg_unmap if mapping isn't yet established */
1657         ASSERT(seg->s_data != NULL);

1659         /* Unmap the whole mapping */
1660 #ifdef DEBUG
1661         ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1662         ASSERT(ret == 0);
1663 #else
1664         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1665 #endif /* DEBUG */
1666 }

1668 /*
1669  * Free the segment from its associated as. This should only be called
1670  * if a mapping to the segment has not yet been established (e.g., if
1671  * an error occurs in the middle of doing an as_map when the segment
1672  * has already been partially set up) or if it has already been deleted
1673  * (e.g., from a segment driver unmap routine if the unmap applies to the
1674  * entire segment). If the mapping is currently set up then seg_unmap() should
1675  * be called instead.
1676  */
1677 void
1678 seg_free(struct seg *seg)
1679 {
1680         register struct as *as = seg->s_as;
1681         struct seg *tseg = as_removeseg(as, seg);

1683         ASSERT(tseg == seg);

1685         /*
1686          * If the segment private data field is NULL,
1687          * then segment driver is not attached yet.
1688          */
1689         if (seg->s_data != NULL)
1690                 SEGOP_FREE(seg);

1692         mutex_destroy(&seg->s_pmtx);
1693         ASSERT(seg->s_phead.p_lnext == &seg->s_phead);
1694         ASSERT(seg->s_phead.p_lprev == &seg->s_phead);
1695         kmem_cache_free(seg_cache, seg);
1696 }

1698 /*ARGSUSED*/
1699 static void
1700 seg_p_mem_config_post_add(
1701         void *arg,
1702         pgcnt_t delta_pages)
1703 {
1704         /* Nothing to do. */
1705 }

1707 void
1708 seg_p_enable(void)
1709 {
1710         mutex_enter(&seg_pcache_mtx);
1711         ASSERT(seg_pdisabled != 0);
```

```
1712         seg_pdisabled--;
1713         mutex_exit(&seg_pcache_mtx);
1714 }

1716 /*
1717  * seg_p_disable - disables seg_pcache, and then attempts to empty the
1718  * cache.
1719  * Returns SEGP_SUCCESS if the cache was successfully emptied, or
1720  * SEGP_FAIL if the cache could not be emptied.
1721  */
1722 int
1723 seg_p_disable(void)
1724 {
1725         pgcnt_t old_plocked;
1726         int stall_count = 0;

1728         mutex_enter(&seg_pcache_mtx);
1729         seg_pdisabled++;
1730         ASSERT(seg_pdisabled != 0);
1731         mutex_exit(&seg_pcache_mtx);

1733         /*
1734          * Attempt to empty the cache. Terminate if seg_plocked does not
1735          * diminish with SEGP_STALL_THRESHOLD consecutive attempts.
1736          */
1737         while (seg_plocked != 0) {
1738                 ASSERT(seg_phashsize_win != 0);
1739                 old_plocked = seg_plocked;
1740                 seg_ppurge_async(1);
1741                 if (seg_plocked == old_plocked) {
1742                         if (stall_count++ > SEGP_STALL_THRESHOLD) {
1743                                 return (SEGP_FAIL);
1744                         }
1745                 } else
1746                         stall_count = 0;
1747                 if (seg_plocked != 0)
1748                         delay(hz/SEGP_PREDEL_DELAY_FACTOR);
1749         }
1750         return (SEGP_SUCCESS);
1751 }

1753 /*
1754  * Attempt to purge seg_pcache.  May need to return before this has
1755  * completed to allow other pre_del callbacks to unlock pages. This is
1756  * ok because:
1757  *      1) The seg_pdisabled flag has been set so at least we won't
1758  *      cache anymore locks and the locks we couldn't purge
1759  *      will not be held if they do get released by a subsequent
1760  *      pre-delete callback.
1761  *
1762  *      2) The rest of the memory delete thread processing does not
1763  *      depend on the changes made in this pre-delete callback. No
1764  *      panics will result, the worst that will happen is that the
1765  *      DR code will timeout and cancel the delete.
1766  */
1767 /*ARGSUSED*/
1768 static int
1769 seg_p_mem_config_pre_del(
1770         void *arg,
1771         pgcnt_t delta_pages)
1772 {
1773         if (seg_phashsize_win == 0) {
1774                 return (0);
1775         }
1776         if (seg_p_disable() != SEGP_SUCCESS)
1777                 cmn_err(CE_NOTE,
```



```
1778                     "!Pre-delete couldn't purge"" pagelock cache - continuing");
1779         return (0);
1780 }

1782 /*ARGSUSED*/
1783 static void
1784 seg_p_mem_config_post_del(
1785         void *arg,
1786         pgcnt_t delta_pages,
1787         int cancelled)
1788 {
1789         if (seg_phashsize_win == 0) {
1790                 return;
1791         }
1792         seg_p_enable();
1793 }

1795 static kphysm_setup_vector_t seg_p_mem_config_vec = {
1796         KPHYSM_SETUP_VECTOR_VERSION,
1797         seg_p_mem_config_post_add,
1798         seg_p_mem_config_pre_del,
1799         seg_p_mem_config_post_del,
1800 };

1802 static void
1803 seg_pinit_mem_config(void)
1804 {
1805         int ret;

1807         ret = kphysm_setup_func_register(&seg_p_mem_config_vec, (void *)NULL);
1808         /*
1809          * Want to catch this in the debug kernel. At run time, if the
1810          * callbacks don't get run all will be OK as the disable just makes
1811          * it more likely that the pages can be collected.
1812          */
1813         ASSERT(ret == 0);
1814 }

1816 /*
1817  * Verify that segment is not a shared anonymous segment which reserves
1818  * swap.  zone.max-swap accounting (zone->zone_max_swap) cannot be transfered
1819  * from one zone to another if any segments are shared.  This is because the
1820  * last process to exit will credit the swap reservation.  This could lead
1821  * to the swap being reserved by one zone, and credited to another.
1822  */
1823 boolean_t
1824 seg_can_change_zones(struct seg *seg)
1825 {
1826         struct segvn_data *svd;

1828         if (seg->s_ops == &segspt_shmops)
1829                 return (B_FALSE);

1831         if (seg->s_ops == &segvn_ops) {
1832                 svd = (struct segvn_data *)seg->s_data;
1833                 if (svd->type == MAP_SHARED &&
1834                     svd->amp != NULL &&
1835                     svd->amp->swresv > 0)
1836                 return (B_FALSE);
1837         }
1838         return (B_TRUE);
1839 }

1841 /*
1842  * Return swap reserved by a segment backing a private mapping.
1843  */
```

```
1844 size_t
1845 seg_swresv(struct seg *seg)
1846 {
1847         struct segvn_data *svd;
1848         size_t swap = 0;

1850         if (seg->s_ops == &segvn_ops) {
1851                 svd = (struct segvn_data *)seg->s_data;
1852                 if (svd->type == MAP_PRIVATE && svd->swresv > 0)
1853                         swap = svd->swresv;
1854         }
1855         return (swap);
1856 }

1858 /*
1859  * General not supported function for SEGOP_INHERIT
1860  */
1861 /* ARGSUSED */
1862 int
1863 seg_inherit_notsup(struct seg *seg, caddr_t addr, size_t len, uint_t op)
1864 {
1865         return (ENOTSUP);
1866 }

1868 /*
1869  * segop wrappers
1870  */
1871 int
1872 segop_dup(struct seg *seg, struct seg *new)
1873 {
1874         return (seg->s_ops->dup(seg, new));
1875 }

1877 int
1878 segop_unmap(struct seg *seg, caddr_t addr, size_t len)
1879 {
1880         return (seg->s_ops->unmap(seg, addr, len));
1881 }

1883 void
1884 segop_free(struct seg *seg)
1885 {
1886         seg->s_ops->free(seg);
1887 }

1889 faultcode_t
1890 segop_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
1891     enum fault_type type, enum seg_rw rw)
1892 {
1893         return (seg->s_ops->fault(hat, seg, addr, len, type, rw));
1894 }

1896 faultcode_t
1897 segop_faulta(struct seg *seg, caddr_t addr)
1898 {
1899         return (seg->s_ops->faulta(seg, addr));
1900 }

1902 int
1903 segop_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
1904 {
1905         return (seg->s_ops->setprot(seg, addr, len, prot));
1906 }

1908 int
1909 segop_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
```



```
1910 {
1911         return (seg->s_ops->checkprot(seg, addr, len, prot));
1912 }

1914 int
1915 segop_kluster(struct seg *seg, caddr_t addr, ssize_t d)
1916 {
1917         return (seg->s_ops->kluster(seg, addr, d));
1918 }

1920 size_t
1921 segop_swapout(struct seg *seg)
1922 {
1923         return (seg->s_ops->swapout(seg));
1924 }

1926 int
1927 segop_sync(struct seg *seg, caddr_t addr, size_t len, int atr, uint_t f)
1928 {
1929         return (seg->s_ops->sync(seg, addr, len, atr, f));
1930 }

1932 size_t
1933 segop_incore(struct seg *seg, caddr_t addr, size_t len, char *v)
1934 {
1935         return (seg->s_ops->incore(seg, addr, len, v));
1936 }

1938 int
1939 segop_lockop(struct seg *seg, caddr_t addr, size_t len, int atr, int op,
1940     ulong_t *b, size_t p)
1941 {
1942         return (seg->s_ops->lockop(seg, addr, len, atr, op, b, p));
1943 }

1945 int
1946 segop_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *p)
1947 {
1948         return (seg->s_ops->getprot(seg, addr, len, p));
1949 }

1951 u_offset_t
1952 segop_getoffset(struct seg *seg, caddr_t addr)
1953 {
1954         return (seg->s_ops->getoffset(seg, addr));
1955 }

1957 int
1958 segop_gettype(struct seg *seg, caddr_t addr)
1959 {
1960         return (seg->s_ops->gettype(seg, addr));
1961 }

1963 int
1964 segop_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
1965 {
1966         return (seg->s_ops->getvp(seg, addr, vpp));
1967 }

1969 int
1970 segop_advise(struct seg *seg, caddr_t addr, size_t len, uint_t b)
1971 {
1972         return (seg->s_ops->advise(seg, addr, len, b));
1973 }

1975 void
```

```
1976 segop_dump(struct seg *seg)
1977 {
1978         seg->s_ops->dump(seg);
1979 }

1981 int
1982 segop_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***page,
1983     enum lock_type type, enum seg_rw rw)
1984 {
1985         return (seg->s_ops->pagelock(seg, addr, len, page, type, rw));
1986 }

1988 int
1989 segop_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
1990 {
1991         return (seg->s_ops->setpagesize(seg, addr, len, szc));
1992 }

1994 int
1995 segop_getmemid(struct seg *seg, caddr_t addr, memid_t *mp)
1996 {
1997         return (seg->s_ops->getmemid(seg, addr, mp));
1998 }

2000 struct lgrp_mem_policy_info *
2001 segop_getpolicy(struct seg *seg, caddr_t addr)
2002 {
2003         if (seg->s_ops->getpolicy == NULL)
2004                 return (NULL);

2006         return (seg->s_ops->getpolicy(seg, addr));
2007 }

2009 int
2010 segop_capable(struct seg *seg, segcapability_t cap)
2011 {
2012         return (seg->s_ops->capable(seg, cap));
2013 }

2015 int
2016 segop_inherit(struct seg *seg, caddr_t addr, size_t len, uint_t op)
2017 {
2018         if (seg->s_ops->inherit == NULL)
2019                 return (ENOTSUP);

2021         return (seg->s_ops->inherit(seg, addr, len, op));
2022 #endif /* ! codereview */
2023 }
```