```
**********************************************************
   54157 Fri May  8 18:05:17 2015
new/usr/src/uts/common/vm/vm_seg.c
patch segpcache-maxwindow-is-useless
**********************************************************
______unchanged_portion_omitted_

 113 /*
 114  * A parameter to control a maximum number of bytes that can be
 115  * purged from pcache at a time.
 116  */
 117 #define P_MAX_APURGE_BYTES      (1024 * 1024 * 1024)

 119 /*
 120  * log2(fraction of pcache to reclaim at a time).
 121  */
 122 #define P_SHRINK_SHFT           (5)

 124 /*
 125  * The following variables can be tuned via /etc/system.
 126  */

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

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

 142 #pragma align 64(pctrl1)
 143 #pragma align 64(pctrl2)
 144 #pragma align 64(pctrl3)

 146 /*
 147  * Keep frequently used variables together in one cache line.
 148  */
 149 static struct p_ctrl1 {
 150         uint_t p_disabled;              /* if not 0, caching temporarily off */
 152         pgcnt_t p_maxwin;               /* max # of pages that can be cached */
 151         size_t p_hashwin_sz;            /* # of non wired buckets */
 152         struct seg_phash *p_htabwin;    /* hash table for non wired entries */
 153         size_t p_hashwired_sz;          /* # of wired buckets */
 154         struct seg_phash_wired *p_htabwired; /* hash table for wired entries */
 155         kmem_cache_t *p_kmcache;        /* kmem cache for seg_pcache structs */
 156 #ifdef _LP64
 157         ulong_t pad[2];
 159         ulong_t pad[1];
 158 #endif /* _LP64 */
 159 } pctrl1;
______unchanged_portion_omitted_

 181 #define seg_pdisabled                   pctrl1.p_disabled
 184 #define seg_pmaxwindow                  pctrl1.p_maxwin
 182 #define seg_phashsize_win               pctrl1.p_hashwin_sz
 183 #define seg_phashtab_win                pctrl1.p_htabwin
 184 #define seg_phashsize_wired             pctrl1.p_hashwired_sz
 185 #define seg_phashtab_wired              pctrl1.p_htabwired
 186 #define seg_pkmcache                    pctrl1.p_kmcache
```



```
 187 #define seg_pmem_mtx                    pctrl2.p_mem_mtx
 188 #define seg_plocked_window              pctrl2.p_locked_win
 189 #define seg_plocked                     pctrl2.p_locked
 190 #define seg_pahcur                      pctrl2.p_ahcur
 191 #define seg_pathr_on                    pctrl2.p_athr_on
 192 #define seg_pahhead                     pctrl2.p_ahhead
 193 #define seg_pmax_pcpage                 pctrl3.p_pcp_maxage
 194 #define seg_pathr_empty_ahb             pctrl3.p_athr_empty_ahb
 195 #define seg_pathr_full_ahb              pctrl3.p_athr_full_ahb
 196 #define seg_pshrink_shift               pctrl3.p_shrink_shft
 197 #define seg_pmaxapurge_npages           pctrl3.p_maxapurge_npages

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

 203 kthread_t *seg_pasync_thr;

 205 extern const struct seg_ops segvn_ops;
 206 extern const struct seg_ops segspt_shmops;

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

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

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

 215 /*
 216  * htag0 argument can be a seg or amp pointer.
 217  */
 218 #define P_HASHBP(seg, htag0, addr, flags)                               \
 219         (IS_PFLAGS_WIRED((flags)) ?                                     \
 220             ((struct seg_phash *)&seg_phashtab_wired[P_HASHWIRED_MASK & \
 221             ((uintptr_t)(htag0) >> P_BASESHIFT)]) :                     \
 222             (&seg_phashtab_win[P_HASHWIN_MASK &                         \
 223             (((uintptr_t)(htag0) >> 3) ^                                \
 224             ((uintptr_t)(addr) >> ((flags & SEGP_PSHIFT) ?              \
 225             (flags >> 16) : page_get_shift((seg)->s_szc))))]))

 227 /*
 228  * htag0 argument can be a seg or amp pointer.
 229  */
 230 #define P_MATCH(pcp, htag0, addr, len)                                  \
 231         ((pcp)->p_htag0 == (htag0) &&                                   \
 232         (pcp)->p_addr == (addr) &&                                      \
 233         (pcp)->p_len >= (len))

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

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

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

 247 /*
 248  * seg_padd_abuck()/seg_premove_abuck() link and unlink hash buckets from
 249  * active hash bucket lists. We maintain active bucket lists to reduce the
 250  * overhead of finding active buckets during asynchronous purging since there
 251  * can be 10s of millions of buckets on a large system but only a small subset
 252  * of them in actual use.
```

```
 253  *
 254  * There’re 2 active bucket lists. Current active list (as per seg_pahcur) is
 255  * used by seg_pinsert()/seg_pinactive()/seg_ppurge() to add and delete
 256  * buckets. The other list is used by asynchronous purge thread. This allows
 257  * the purge thread to walk its active list without holding seg_pmem_mtx for a
 258  * long time. When asynchronous thread is done with its list it switches to
 259  * current active list and makes the list it just finished processing as
 260  * current active list.
 261  *
 262  * seg_padd_abuck() only adds the bucket to current list if the bucket is not
 263  * yet on any list.  seg_premove_abuck() may remove the bucket from either
 264  * list. If the bucket is on current list it will be always removed. Otherwise
 265  * the bucket is only removed if asynchronous purge thread is not currently
 266  * running or seg_premove_abuck() is called by asynchronous purge thread
 267  * itself. A given bucket can only be on one of active lists at a time. These
 268  * routines should be called with per bucket lock held.  The routines use
 269  * seg_pmem_mtx to protect list updates. seg_padd_abuck() must be called after
 270  * the first entry is added to the bucket chain and seg_premove_abuck() must
 271  * be called after the last pcp entry is deleted from its chain. Per bucket
 272  * lock should be held by the callers.  This avoids a potential race condition
 273  * when seg_premove_abuck() removes a bucket after pcp entries are added to
 274  * its list after the caller checked that the bucket has no entries. (this
 275  * race would cause a loss of an active bucket from the active lists).
 276  *
 277  * Both lists are circular doubly linked lists anchored at seg_pahhead heads.
 278  * New entries are added to the end of the list since LRU is used as the
 279  * purging policy.
 280  */
 281 static void
 282 seg_padd_abuck(struct seg_phash *hp)
 283 {
 284         int lix;

 286         ASSERT(MUTEX_HELD(&hp->p_hmutex));
 287         ASSERT((struct seg_phash *)hp->p_hnext != hp);
 288         ASSERT((struct seg_phash *)hp->p_hprev != hp);
 289         ASSERT(hp->p_hnext == hp->p_hprev);
 290         ASSERT(!IS_PCP_WIRED(hp->p_hnext));
 291         ASSERT(hp->p_hnext->p_hnext == (struct seg_pcache *)hp);
 292         ASSERT(hp->p_hprev->p_hprev == (struct seg_pcache *)hp);
 293         ASSERT(hp >= seg_phashtab_win &&
 294             hp < &seg_phashtab_win[seg_phashsize_win]);

 296         /*
 297          * This bucket can already be on one of active lists
 298          * since seg_premove_abuck() may have failed to remove it
 299          * before.
 300          */
 301         mutex_enter(&seg_pmem_mtx);
 302         lix = seg_pahcur;
 303         ASSERT(lix >= 0 && lix <= 1);
 304         if (hp->p_halink[lix].p_lnext != NULL) {
 305                 ASSERT(hp->p_halink[lix].p_lprev != NULL);
 306                 ASSERT(hp->p_halink[!lix].p_lnext == NULL);
 307                 ASSERT(hp->p_halink[!lix].p_lprev == NULL);
 308                 mutex_exit(&seg_pmem_mtx);
 309                 return;
 310         }
 311         ASSERT(hp->p_halink[lix].p_lprev == NULL);

 313         /*
 314          * If this bucket is still on list !lix async thread can’t yet remove
 315          * it since we hold here per bucket lock. In this case just return
 316          * since async thread will eventually find and process this bucket.
 317          */
 318         if (hp->p_halink[!lix].p_lnext != NULL) {
```



```
 319                 ASSERT(hp->p_halink[!lix].p_lprev != NULL);
 320                 mutex_exit(&seg_pmem_mtx);
 321                 return;
 322         }
 323         ASSERT(hp->p_halink[!lix].p_lprev == NULL);
 324         /*
 325          * This bucket is not on any active bucket list yet.
 326          * Add the bucket to the tail of current active list.
 327          */
 328         hp->p_halink[lix].p_lnext = &seg_pahhead[lix];
 329         hp->p_halink[lix].p_lprev = seg_pahhead[lix].p_lprev;
 330         seg_pahhead[lix].p_lprev->p_lnext = &hp->p_halink[lix];
 331         seg_pahhead[lix].p_lprev = &hp->p_halink[lix];
 332         mutex_exit(&seg_pmem_mtx);
 333 }
______unchanged_portion_omitted_

 726 #ifdef DEBUG
 727 static uint32_t p_insert_chk_mtbf = 0;
 728 #endif

 730 /*
 731  * The seg_pinsert_check() is used by segment drivers to predict whether
 732  * a call to seg_pinsert will fail and thereby avoid wasteful pre-processing.
 733  */
 734 /*ARGSUSED*/
 735 int
 736 seg_pinsert_check(struct seg *seg, struct anon_map *amp, caddr_t addr,
 737     size_t len, uint_t flags)
 738 {
 739         ASSERT(seg != NULL);

 741 #ifdef DEBUG
 742         if (p_insert_chk_mtbf && !(gethrtime() % p_insert_chk_mtbf)) {
 743                 return (SEGP_FAIL);
 744         }
 745 #endif

 747         if (seg_pdisabled) {
 748                 return (SEGP_FAIL);
 749         }
 750         ASSERT(seg_phashsize_win != 0);

 752         if (IS_PFLAGS_WIRED(flags)) {
 753                 return (SEGP_SUCCESS);
 754         }

 759         if (seg_plocked_window + btop(len) > seg_pmaxwindow) {
 760                 return (SEGP_FAIL);
 761         }

 756         if (freemem < desfree) {
 757                 return (SEGP_FAIL);
 758         }

 760         return (SEGP_SUCCESS);
 761 }

 763 #ifdef DEBUG
 764 static uint32_t p_insert_mtbf = 0;
 765 #endif

 767 /*
 768  * Insert address range with shadow list into pagelock cache if there’s no
 769  * shadow list already cached for this address range. If the cache is off or
 770  * caching is temporarily disabled or the allowed ’window’ is exceeded return
```

```
 771  * SEGP_FAIL. Otherwise return SEGP_SUCCESS.
 772  *
 773  * For non wired shadow lists (segvn case) include address in the hashing
 774  * function to avoid linking all the entries from the same segment or amp on
 775  * the same bucket.  amp is used instead of seg if amp is not NULL. Non wired
 776  * pcache entries are also linked on a per segment/amp list so that all
 777  * entries can be found quickly during seg/amp purge without walking the
 778  * entire pcache hash table.  For wired shadow lists (segspt case) we
 779  * don't use address hashing and per segment linking because the caller
 780  * currently inserts only one entry per segment that covers the entire
 781  * segment. If we used per segment linking even for segspt it would complicate
 782  * seg_ppurge_wiredpp() locking.
 783  *
 784  * Both hash bucket and per seg/amp locks need to be held before adding a non
 785  * wired entry to hash and per seg/amp lists. per seg/amp lock should be taken
 786  * first.
 787  *
 788  * This function will also remove from pcache old inactive shadow lists that
 789  * overlap with this request but cover smaller range for the same start
 790  * address.
 791  */
 792 int
 793 seg_pinsert(struct seg *seg, struct anon_map *amp, caddr_t addr, size_t len,
 794     size_t wlen, struct page **pp, enum seg_rw rw, uint_t flags,
 795     seg_preclaim_cbfunc_t callback)
 796 {
 797         struct seg_pcache *pcp;
 798         struct seg_phash *hp;
 799         pgcnt_t npages;
 800         pcache_link_t *pheadp;
 801         kmutex_t *pmtx;
 802         struct seg_pcache *delcallb_list = NULL;

 804         ASSERT(seg != NULL);
 805         ASSERT(rw == S_READ || rw == S_WRITE);
 806         ASSERT(rw == S_READ || wlen == len);
 807         ASSERT(rw == S_WRITE || wlen <= len);
 808         ASSERT(amp == NULL || wlen == len);

 810 #ifdef DEBUG
 811         if (p_insert_mtbf && !(gethrtime() % p_insert_mtbf)) {
 812                 return (SEGP_FAIL);
 813         }
 814 #endif

 816         if (seg_pdisabled) {
 817                 return (SEGP_FAIL);
 818         }
 819         ASSERT(seg_phashsize_win != 0);

 821         ASSERT((len & PAGEOFFSET) == 0);
 822         npages = btop(len);
 823         mutex_enter(&seg_pmem_mtx);
 824         if (!IS_PFLAGS_WIRED(flags)) {
 832                 if (seg_plocked_window + npages > seg_pmaxwindow) {
 833                         mutex_exit(&seg_pmem_mtx);
 834                         return (SEGP_FAIL);
 835                 }
 825                 seg_plocked_window += npages;
 826         }
 827         seg_plocked += npages;
 828         mutex_exit(&seg_pmem_mtx);

 830         pcp = kmem_cache_alloc(seg_pkmcache, KM_SLEEP);
 831         /*
 832          * If amp is not NULL set htag0 to amp otherwise set it to seg.
```



```
 833          */
 834         if (amp == NULL) {
 835                 pcp->p_htag0 = (void *)seg;
 836                 pcp->p_flags = flags & 0xffff;
 837         } else {
 838                 pcp->p_htag0 = (void *)amp;
 839                 pcp->p_flags = (flags & 0xffff) | SEGP_AMP;
 840         }
 841         pcp->p_addr = addr;
 842         pcp->p_len = len;
 843         pcp->p_wlen = wlen;
 844         pcp->p_pp = pp;
 845         pcp->p_write = (rw == S_WRITE);
 846         pcp->p_callback = callback;
 847         pcp->p_active = 1;

 849         hp = P_HASHBP(seg, pcp->p_htag0, addr, flags);
 850         if (!IS_PFLAGS_WIRED(flags)) {
 851                 int found;
 852                 void *htag0;
 853                 if (amp == NULL) {
 854                         pheadp = &seg->s_phead;
 855                         pmtx = &seg->s_pmtx;
 856                         htag0 = (void *)seg;
 857                 } else {
 858                         pheadp = &amp->a_phead;
 859                         pmtx = &amp->a_pmtx;
 860                         htag0 = (void *)amp;
 861                 }
 862                 mutex_enter(pmtx);
 863                 mutex_enter(&hp->p_hmutex);
 864                 delcallb_list = seg_plookup_checkdup(hp, htag0, addr,
 865                     len, &found);
 866                 if (found) {
 867                         mutex_exit(&hp->p_hmutex);
 868                         mutex_exit(pmtx);
 869                         mutex_enter(&seg_pmem_mtx);
 870                         seg_plocked -= npages;
 871                         seg_plocked_window -= npages;
 872                         mutex_exit(&seg_pmem_mtx);
 873                         kmem_cache_free(seg_pkmcache, pcp);
 874                         goto out;
 875                 }
 876                 pcp->p_plink.p_lnext = pheadp->p_lnext;
 877                 pcp->p_plink.p_lprev = pheadp;
 878                 pheadp->p_lnext->p_lprev = &pcp->p_plink;
 879                 pheadp->p_lnext = &pcp->p_plink;
 880         } else {
 881                 mutex_enter(&hp->p_hmutex);
 882         }
 883         pcp->p_hashp = hp;
 884         pcp->p_hnext = hp->p_hnext;
 885         pcp->p_hprev = (struct seg_pcache *)hp;
 886         hp->p_hnext->p_hprev = pcp;
 887         hp->p_hnext = pcp;
 888         if (!IS_PFLAGS_WIRED(flags) &&
 889             hp->p_hprev == pcp) {
 890                 seg_padd_abuck(hp);
 891         }
 892         mutex_exit(&hp->p_hmutex);
 893         if (!IS_PFLAGS_WIRED(flags)) {
 894                 mutex_exit(pmtx);
 895         }

 897 out:
 898         npages = 0;
```

```
899         while (delcallb_list != NULL) {
900                 pcp = delcallb_list;
901                 delcallb_list = pcp->p_hprev;
902                 ASSERT(!IS_PCP_WIRED(pcp) && !pcp->p_active);
903                 (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
904                     pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 0);
905                 npages += btop(pcp->p_len);
906                 kmem_cache_free(seg_pkmcache, pcp);
907         }
908         if (npages) {
909                 ASSERT(!IS_PFLAGS_WIRED(flags));
910                 mutex_enter(&seg_pmem_mtx);
911                 ASSERT(seg_plocked >= npages);
912                 ASSERT(seg_plocked_window >= npages);
913                 seg_plocked -= npages;
914                 seg_plocked_window -= npages;
915                 mutex_exit(&seg_pmem_mtx);
916         }

918         return (SEGP_SUCCESS);
919 }

921 /*
922  * purge entries from the pagelock cache if not active
923  * and not recently used.
924  */
925 static void
926 seg_ppurge_async(int force)
927 {
928         struct seg_pcache *delcallb_list = NULL;
929         struct seg_pcache *pcp;
930         struct seg_phash *hp;
931         pgcnt_t npages = 0;
932         pgcnt_t npages_window = 0;
933         pgcnt_t npgs_to_purge;
934         pgcnt_t npgs_purged = 0;
935         int hlinks = 0;
936         int hlix;
937         pcache_link_t *hlinkp;
938         pcache_link_t *hlnextp = NULL;
939         int lowmem;
951         int trim;

941         ASSERT(seg_phashsize_win != 0);

943         /*
944          * if the cache is off or empty, return
945          */
946         if (seg_plocked == 0 || (!force && seg_plocked_window == 0)) {
947                 return;
948         }

950         if (!force) {
951                 lowmem = 0;
964                 trim = 0;
952                 if (freemem < lotsfree + needfree) {
953                         spgcnt_t fmem = MAX((spgcnt_t)(freemem - needfree), 0);
954                         if (fmem <= 5 * (desfree >> 2)) {
955                                 lowmem = 1;
956                         } else if (fmem <= 7 * (lotsfree >> 3)) {
957                                 if (seg_plocked_window >=
958                                     (availrmem_initial >> 1)) {
959                                         lowmem = 1;
960                                 }
961                         } else if (fmem < lotsfree) {
962                                 if (seg_plocked_window >=
```



```
963                                     3 * (availrmem_initial >> 2)) {
964                                         lowmem = 1;
965                                 }
966                         }
967                 }
968                 if (!lowmem) {
981                 if (seg_plocked_window >= 7 * (seg_pmaxwindow >> 3)) {
982                         trim = 1;
983                 }
984                 if (!lowmem && !trim) {
969                         return;
970                 }
971                 npgs_to_purge = seg_plocked_window >>
972                     seg_pshrink_shift;
973                 if (lowmem) {
974                         npgs_to_purge = MIN(npgs_to_purge,
975                             MAX(seg_pmaxapurge_npages, desfree));
976                 } else {
977                         npgs_to_purge = MIN(npgs_to_purge,
978                             seg_pmaxapurge_npages);
979                 }
980                 if (npgs_to_purge == 0) {
981                         return;
982                 }
983         } else {
984                 struct seg_phash_wired *hpw;

986                 ASSERT(seg_phashsize_wired != 0);

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

991                         if (hpw->p_hnext == (struct seg_pcache *)hpw) {
992                                 continue;
993                         }

995                         mutex_enter(&hpw->p_hmutex);

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

1001                                ASSERT(IS_PCP_WIRED(pcp));
1002                                ASSERT(pcp->p_hashp ==
1003                                    (struct seg_phash *)hpw);

1005                                if (pcp->p_active) {
1006                                        continue;
1007                                }
1008                                pcp->p_hprev->p_hnext = pcp->p_hnext;
1009                                pcp->p_hnext->p_hprev = pcp->p_hprev;
1010                                pcp->p_hprev = delcallb_list;
1011                                delcallb_list = pcp;
1012                        }
1013                        mutex_exit(&hpw->p_hmutex);
1014                }
1015        }

1017        mutex_enter(&seg_pmem_mtx);
1018        if (seg_pathr_on) {
1019                mutex_exit(&seg_pmem_mtx);
1020                goto runcb;
1021        }
1022        seg_pathr_on = 1;
1023        mutex_exit(&seg_pmem_mtx);
1024        ASSERT(seg_pahcur <= 1);
```

```
1025         hlix = !seg_pahcur;

1027 again:
1028         for (hlinkp = seg_pahhead[hlix].p_lnext; hlinkp != &seg_pahhead[hlix];
1029             hlinkp = hlnextp) {

1031                 hlnextp = hlinkp->p_lnext;
1032                 ASSERT(hlnextp != NULL);

1034                 hp = hlink2phash(hlinkp, hlix);
1035                 if (hp->p_hnext == (struct seg_pcache *)hp) {
1036                         seg_pathr_empty_ahb++;
1037                         continue;
1038                 }
1039                 seg_pathr_full_ahb++;
1040                 mutex_enter(&hp->p_hmutex);

1042                 for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
1043                     pcp = pcp->p_hnext) {
1044                         pcache_link_t *pheadp;
1045                         pcache_link_t *plinkp;
1046                         void *htag0;
1047                         kmutex_t *pmtx;

1049                         ASSERT(!IS_PCP_WIRED(pcp));
1050                         ASSERT(pcp->p_hashp == hp);

1052                         if (pcp->p_active) {
1053                                 continue;
1054                         }
1055                         if (!force && pcp->p_ref &&
1056                             PCP_AGE(pcp) < seg_pmax_pcpage) {
1057                                 pcp->p_ref = 0;
1058                                 continue;
1059                         }
1060                         plinkp = &pcp->p_plink;
1061                         htag0 = pcp->p_htag0;
1062                         if (pcp->p_flags & SEGP_AMP) {
1063                                 pheadp = &((amp_t *)htag0)->a_phead;
1064                                 pmtx = &((amp_t *)htag0)->a_pmtx;
1065                         } else {
1066                                 pheadp = &((seg_t *)htag0)->s_phead;
1067                                 pmtx = &((seg_t *)htag0)->s_pmtx;
1068                         }
1069                         if (!mutex_tryenter(pmtx)) {
1070                                 continue;
1071                         }
1072                         ASSERT(pheadp->p_lnext != pheadp);
1073                         ASSERT(pheadp->p_lprev != pheadp);
1074                         plinkp->p_lprev->p_lnext =
1075                             plinkp->p_lnext;
1076                         plinkp->p_lnext->p_lprev =
1077                             plinkp->p_lprev;
1078                         pcp->p_hprev->p_hnext = pcp->p_hnext;
1079                         pcp->p_hnext->p_hprev = pcp->p_hprev;
1080                         mutex_exit(pmtx);
1081                         pcp->p_hprev = delcallb_list;
1082                         delcallb_list = pcp;
1083                         npgs_purged += btop(pcp->p_len);
1084                 }
1085                 if (hp->p_hnext == (struct seg_pcache *)hp) {
1086                         seg_premove_abuck(hp, 1);
1087                 }
1088                 mutex_exit(&hp->p_hmutex);
1089                 if (npgs_purged >= seg_plocked_window) {
1090                         break;
```



```
1091                 }
1092                 if (!force) {
1093                         if (npgs_purged >= npgs_to_purge) {
1094                                 break;
1095                         }
1096                         if (!(seg_pathr_full_ahb & 15)) {
1112                         if (!trim && !(seg_pathr_full_ahb & 15)) {
1097                                 ASSERT(lowmem);
1098                                 if (freemem >= lotsfree + needfree) {
1099                                         break;
1100                                 }
1101                         }
1102                 }
1103         }

1105         if (hlinkp == &seg_pahhead[hlix]) {
1106                 /*
1107                  * We processed the entire hlix active bucket list
1108                  * but didn't find enough pages to reclaim.
1109                  * Switch the lists and walk the other list
1110                  * if we haven't done it yet.
1111                  */
1112                 mutex_enter(&seg_pmem_mtx);
1113                 ASSERT(seg_pathr_on);
1114                 ASSERT(seg_pahcur == !hlix);
1115                 seg_pahcur = hlix;
1116                 mutex_exit(&seg_pmem_mtx);
1117                 if (++hlinks < 2) {
1118                         hlix = !hlix;
1119                         goto again;
1120                 }
1121         } else if ((hlinkp = hlnextp) != &seg_pahhead[hlix] &&
1122             seg_pahhead[hlix].p_lnext != hlinkp) {
1123                 ASSERT(hlinkp != NULL);
1124                 ASSERT(hlinkp->p_lprev != &seg_pahhead[hlix]);
1125                 ASSERT(seg_pahhead[hlix].p_lnext != &seg_pahhead[hlix]);
1126                 ASSERT(seg_pahhead[hlix].p_lprev != &seg_pahhead[hlix]);

1128                 /*
1129                  * Reinsert the header to point to hlinkp
1130                  * so that we start from hlinkp bucket next time around.
1131                  */
1132                 seg_pahhead[hlix].p_lnext->p_lprev = seg_pahhead[hlix].p_lprev;
1133                 seg_pahhead[hlix].p_lprev->p_lnext = seg_pahhead[hlix].p_lnext;
1134                 seg_pahhead[hlix].p_lnext = hlinkp;
1135                 seg_pahhead[hlix].p_lprev = hlinkp->p_lprev;
1136                 hlinkp->p_lprev->p_lnext = &seg_pahhead[hlix];
1137                 hlinkp->p_lprev = &seg_pahhead[hlix];
1138         }

1140         mutex_enter(&seg_pmem_mtx);
1141         ASSERT(seg_pathr_on);
1142         seg_pathr_on = 0;
1143         mutex_exit(&seg_pmem_mtx);

1145 runcb:
1146         /*
1147          * Run the delayed callback list. segments/amps can't go away until
1148          * callback is executed since they must have non 0 softlockcnt. That's
1149          * why we don't need to hold as/seg/amp locks to execute the callback.
1150          */
1151         while (delcallb_list != NULL) {
1152                 pcp = delcallb_list;
1153                 delcallb_list = pcp->p_hprev;
1154                 ASSERT(!pcp->p_active);
1155                 (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
```

```
1156                        pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 1);
1157                npages += btop(pcp->p_len);
1158                if (!IS_PCP_WIRED(pcp)) {
1159                        npages_window += btop(pcp->p_len);
1160                }
1161                kmem_cache_free(seg_pkmcache, pcp);
1162        }
1163        if (npages) {
1164                mutex_enter(&seg_pmem_mtx);
1165                ASSERT(seg_plocked >= npages);
1166                ASSERT(seg_plocked_window >= npages_window);
1167                seg_plocked -= npages;
1168                seg_plocked_window -= npages_window;
1169                mutex_exit(&seg_pmem_mtx);
1170        }
1171 }
______unchanged_portion_omitted_

1348 static void seg_pinit_mem_config(void);

1350 /*
1351  * setup the pagelock cache
1352  */
1353 static void
1354 seg_pinit(void)
1355 {
1356        struct seg_phash *hp;
1357        ulong_t i;
1358        pgcnt_t physmegs;

1360        seg_plocked = 0;
1361        seg_plocked_window = 0;

1363        if (segpcache_enabled == 0) {
1364                seg_phashsize_win = 0;
1365                seg_phashsize_wired = 0;
1366                seg_pdisabled = 1;
1367                return;
1368        }

1370        seg_pdisabled = 0;
1371        seg_pkmcache = kmem_cache_create("seg_pcache",
1372            sizeof (struct seg_pcache), 0, NULL, NULL, NULL, NULL, NULL, 0);
1373        if (segpcache_pcp_maxage_ticks <= 0) {
1374                segpcache_pcp_maxage_ticks = segpcache_pcp_maxage_sec * hz;
1375        }
1376        seg_pmax_pcpage = segpcache_pcp_maxage_ticks;
1377        seg_pathr_empty_ahb = 0;
1378        seg_pathr_full_ahb = 0;
1379        seg_pshrink_shift = segpcache_shrink_shift;
1380        seg_pmaxapurge_npages = btop(segpcache_maxapurge_bytes);

1382        mutex_init(&seg_pcache_mtx, NULL, MUTEX_DEFAULT, NULL);
1383        mutex_init(&seg_pmem_mtx, NULL, MUTEX_DEFAULT, NULL);
1384        mutex_init(&seg_pasync_mtx, NULL, MUTEX_DEFAULT, NULL);
1385        cv_init(&seg_pasync_cv, NULL, CV_DEFAULT, NULL);

1387        physmegs = physmem >> (20 - PAGESHIFT);

1389        /*
1390         * If segpcache_hashsize_win was not set in /etc/system or it has
1391         * absurd value set it to a default.
1392         */
1393        if (segpcache_hashsize_win == 0 || segpcache_hashsize_win > physmem) {
1394                /*
1395                 * Create one bucket per 32K (or at least per 8 pages) of
```



```
1396                 * available memory.
1397                 */
1398                pgcnt_t pages_per_bucket = MAX(btop(32 * 1024), 8);
1399                segpcache_hashsize_win = MAX(1024, physmem / pages_per_bucket);
1400        }
1401        if (!ISP2(segpcache_hashsize_win)) {
1402                ulong_t rndfac = ~(1UL <<
1403                    (highbit(segpcache_hashsize_win) - 1));
1404                rndfac &= segpcache_hashsize_win;
1405                segpcache_hashsize_win += rndfac;
1406                segpcache_hashsize_win = 1 <<
1407                    (highbit(segpcache_hashsize_win) - 1);
1408        }
1409        seg_phashsize_win = segpcache_hashsize_win;
1410        seg_phashtab_win = kmem_zalloc(
1411            seg_phashsize_win * sizeof (struct seg_phash),
1412            KM_SLEEP);
1413        for (i = 0; i < seg_phashsize_win; i++) {
1414                hp = &seg_phashtab_win[i];
1415                hp->p_hnext = (struct seg_pcache *)hp;
1416                hp->p_hprev = (struct seg_pcache *)hp;
1417                mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1418        }

1420        seg_pahcur = 0;
1421        seg_pathr_on = 0;
1422        seg_pahhead[0].p_lnext = &seg_pahhead[0];
1423        seg_pahhead[0].p_lprev = &seg_pahhead[0];
1424        seg_pahhead[1].p_lnext = &seg_pahhead[1];
1425        seg_pahhead[1].p_lprev = &seg_pahhead[1];

1427        /*
1428         * If segpcache_hashsize_wired was not set in /etc/system or it has
1429         * absurd value set it to a default.
1430         */
1431        if (segpcache_hashsize_wired == 0 ||
1432            segpcache_hashsize_wired > physmem / 4) {
1433                /*
1434                 * Choose segpcache_hashsize_wired based on physmem.
1435                 * Create a bucket per 128K bytes upto 256K buckets.
1436                 */
1437                if (physmegs < 20 * 1024) {
1438                        segpcache_hashsize_wired = MAX(1024, physmegs << 3);
1439                } else {
1440                        segpcache_hashsize_wired = 256 * 1024;
1441                }
1442        }
1443        if (!ISP2(segpcache_hashsize_wired)) {
1444                segpcache_hashsize_wired = 1 <<
1445                    highbit(segpcache_hashsize_wired);
1446        }
1447        seg_phashsize_wired = segpcache_hashsize_wired;
1448        seg_phashtab_wired = kmem_zalloc(
1449            seg_phashsize_wired * sizeof (struct seg_phash_wired), KM_SLEEP);
1450        for (i = 0; i < seg_phashsize_wired; i++) {
1451                hp = (struct seg_phash *)&seg_phashtab_wired[i];
1452                hp->p_hnext = (struct seg_pcache *)hp;
1453                hp->p_hprev = (struct seg_pcache *)hp;
1454                mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1455        }

1473        if (segpcache_maxwindow == 0) {
1474                if (physmegs < 64) {
1475                        /* 3% of memory */
1476                        segpcache_maxwindow = availrmem >> 5;
1477                } else if (physmegs < 512) {
```

```
1478                        /* 12% of memory */
1479                        segpcache_maxwindow = availrmem >> 3;
1480                } else if (physmegs < 1024) {
1481                        /* 25% of memory */
1482                        segpcache_maxwindow = availrmem >> 2;
1483                } else if (physmegs < 2048) {
1484                        /* 50% of memory */
1485                        segpcache_maxwindow = availrmem >> 1;
1486                } else {
1487                        /* no limit */
1488                        segpcache_maxwindow = (pgcnt_t)-1;
1489                }
1490        }
1491        seg_pmaxwindow = segpcache_maxwindow;
1457        seg_pinit_mem_config();
1458 }
```

_____***unchanged_portion_omitted_***