```
**********************************************************
   31976 Thu Jan  8 09:14:29 2015
new/usr/src/uts/common/vm/vm_pvn.c
5384 pvn_getpages may assert in valid scenarios
**********************************************************
```

```
/*
 * 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 (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#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 - paged vnode.
 *
 * This file supplies vm support for the vnode operations that deal with pages.
 */
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/vnode.h>
#include <sys/uio.h>
#include <sys/vmsystm.h>
#include <sys/mman.h>
#include <sys/vfs.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
```



```
#include <sys/cpuvar.h>
#include <sys/vtrace.h>
#include <sys/tnf_probe.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/rm.h>
#include <vm/pvn.h>
#include <vm/page.h>
#include <vm/seg_map.h>
#include <vm/seg_kmem.h>
#include <sys/fs/swapnode.h>

int pvn_nofodklust = 0;
int pvn_write_noklust = 0;

uint_t pvn_vmodsort_supported = 0;      /* set if HAT supports VMODSORT */
uint_t pvn_vmodsort_disable = 0;        /* set in /etc/system to disable HAT */
                                        /* support for vmodsort for testing */

static struct kmem_cache *marker_cache = NULL;

/*
 * Find the largest contiguous block which contains ‘addr’ for file offset
 * ‘offset’ in it while living within the file system block sizes (‘vp_off’
 * and ‘vp_len’) and the address space limits for which no pages currently
 * exist and which map to consecutive file offsets.
 */
page_t *
pvn_read_kluster(
        struct vnode *vp,
        u_offset_t off,
        struct seg *seg,
        caddr_t addr,
        u_offset_t *offp,                       /* return values */
        size_t *lenp,                           /* return values */
        u_offset_t vp_off,
        size_t vp_len,
        int isra)
{
        ssize_t deltaf, deltab;
        page_t *pp;
        page_t *plist = NULL;
        spgcnt_t pagesavail;
        u_offset_t vp_end;

        ASSERT(off >= vp_off && off < vp_off + vp_len);

        /*
         * We only want to do klustering/read ahead if there
         * is more than minfree pages currently available.
         */
        pagesavail = freemem - minfree;

        if (pagesavail <= 0)
                if (isra)
                        return ((page_t *)NULL);    /* ra case - give up */
                else
                        pagesavail = 1;             /* must return a page */

        /* We calculate in pages instead of bytes due to 32-bit overflows */
        if (pagesavail < (spgcnt_t)btopr(vp_len)) {
                /*
                 * Don’t have enough free memory for the
                 * max request, try sizing down vp request.
```

```
128                  */
129                 deltab = (ssize_t)(off - vp_off);
130                 vp_len -= deltab;
131                 vp_off += deltab;
132                 if (pagesavail < btopr(vp_len)) {
133                         /*
134                          * Still not enough memory, just settle for
135                          * pagesavail which is at least 1.
136                          */
137                         vp_len = ptob(pagesavail);
138                 }
139         }

141         vp_end = vp_off + vp_len;
142         ASSERT(off >= vp_off && off < vp_end);

144         if (isra && SEGOP_KLUSTER(seg, addr, 0))
145                 return ((page_t *)NULL);        /* segment driver says no */

147         if ((plist = page_create_va(vp, off,
148             PAGESIZE, PG_EXCL | PG_WAIT, seg, addr)) == NULL)
149                 return ((page_t *)NULL);

151         if (vp_len <= PAGESIZE || pvn_nofodklust) {
152                 *offp = off;
153                 *lenp = MIN(vp_len, PAGESIZE);
154         } else {
155                 /*
156                  * Scan back from front by incrementing "deltab" and
157                  * comparing "off" with "vp_off + deltab" to avoid
158                  * "signed" versus "unsigned" conversion problems.
159                  */
160                 for (deltab = PAGESIZE; off >= vp_off + deltab;
161                     deltab += PAGESIZE) {
162                         /*
163                          * Call back to the segment driver to verify that
164                          * the klustering/read ahead operation makes sense.
165                          */
166                         if (SEGOP_KLUSTER(seg, addr, -deltab))
167                                 break;          /* page not eligible */
168                         if ((pp = page_create_va(vp, off - deltab,
169                             PAGESIZE, PG_EXCL, seg, addr - deltab))
170                             == NULL)
171                                 break;          /* already have the page */
172                         /*
173                          * Add page to front of page list.
174                          */
175                         page_add(&plist, pp);
176                 }
177                 deltab -= PAGESIZE;

179                 /* scan forward from front */
180                 for (deltaf = PAGESIZE; off + deltaf < vp_end;
181                     deltaf += PAGESIZE) {
182                         /*
183                          * Call back to the segment driver to verify that
184                          * the klustering/read ahead operation makes sense.
185                          */
186                         if (SEGOP_KLUSTER(seg, addr, deltaf))
187                                 break;          /* page not file extension */
188                         if ((pp = page_create_va(vp, off + deltaf,
189                             PAGESIZE, PG_EXCL, seg, addr + deltaf))
190                             == NULL)
191                                 break;          /* already have page */

193                         /*
```



```
194                          * Add page to end of page list.
195                          */
196                         page_add(&plist, pp);
197                         plist = plist->p_next;
198                 }
199                 *offp = off = off - deltab;
200                 *lenp = deltab + deltaf;
201                 ASSERT(off >= vp_off);

203                 /*
204                  * If we ended up getting more than was actually
205                  * requested, retract the returned length to only
206                  * reflect what was requested.  This might happen
207                  * if we were allowed to kluster pages across a
208                  * span of (say) 5 frags, and frag size is less
209                  * than PAGESIZE.  We need a whole number of
210                  * pages to contain those frags, but the returned
211                  * size should only allow the returned range to
212                  * extend as far as the end of the frags.
213                  */
214                 if ((vp_off + vp_len) < (off + *lenp)) {
215                         ASSERT(vp_end > off);
216                         *lenp = vp_end - off;
217                 }
218         }
219         TRACE_3(TR_FAC_VM, TR_PVN_READ_KLUSTER,
220             "pvn_read_kluster:seg %p addr %x isra %x",
221             seg, addr, isra);
222         return (plist);
223 }

225 /*
226  * Handle pages for this vnode on either side of the page "pp"
227  * which has been locked by the caller.  This routine will also
228  * do klustering in the range [vp_off, vp_off + vp_len] up
229  * until a page which is not found.  The offset and length
230  * of pages included is returned in "*offp" and "*lenp".
231  *
232  * Returns a list of dirty locked pages all ready to be
233  * written back.
234  */
235 page_t *
236 pvn_write_kluster(
237         struct vnode *vp,
238         page_t *pp,
239         u_offset_t *offp,               /* return values */
240         size_t *lenp,                   /* return values */
241         u_offset_t vp_off,
242         size_t vp_len,
243         int flags)
244 {
245         u_offset_t off;
246         page_t *dirty;
247         size_t deltab, deltaf;
248         se_t se;
249         u_offset_t vp_end;

251         off = pp->p_offset;

253         /*
254          * Kustering should not be done if we are invalidating
255          * pages since we could destroy pages that belong to
256          * some other process if this is a swap vnode.
257          */
258         if (pvn_write_noklust || ((flags & B_INVAL) && IS_SWAPVP(vp))) {
259                 *offp = off;
```

```
260                 *lenp = PAGESIZE;
261                 return (pp);
262         }

264         if (flags & (B_FREE | B_INVAL))
265                 se = SE_EXCL;
266         else
267                 se = SE_SHARED;

269         dirty = pp;
270         /*
271          * Scan backwards looking for pages to kluster by incrementing
272          * "deltab" and comparing "off" with "vp_off + deltab" to
273          * avoid "signed" versus "unsigned" conversion problems.
274          */
275         for (deltab = PAGESIZE; off >= vp_off + deltab; deltab += PAGESIZE) {
276                 pp = page_lookup_nowait(vp, off - deltab, se);
277                 if (pp == NULL)
278                         break;          /* page not found */
279                 if (pvn_getdirty(pp, flags | B_DELWRI) == 0)
280                         break;
281                 page_add(&dirty, pp);
282         }
283         deltab -= PAGESIZE;

285         vp_end = vp_off + vp_len;
286         /* now scan forwards looking for pages to kluster */
287         for (deltaf = PAGESIZE; off + deltaf < vp_end; deltaf += PAGESIZE) {
288                 pp = page_lookup_nowait(vp, off + deltaf, se);
289                 if (pp == NULL)
290                         break;          /* page not found */
291                 if (pvn_getdirty(pp, flags | B_DELWRI) == 0)
292                         break;
293                 page_add(&dirty, pp);
294                 dirty = dirty->p_next;
295         }

297         *offp = off - deltab;
298         *lenp = deltab + deltaf;
299         return (dirty);
300 }

302 /*
303  * Generic entry point used to release the "shared/exclusive" lock
304  * and the "p_iolock" on pages after i/o is complete.
305  */
306 void
307 pvn_io_done(page_t *plist)
308 {
309         page_t *pp;

311         while (plist != NULL) {
312                 pp = plist;
313                 page_sub(&plist, pp);
314                 page_io_unlock(pp);
315                 page_unlock(pp);
316         }
317 }

319 /*
320  * Entry point to be used by file system getpage subr's and
321  * other such routines which either want to unlock pages (B_ASYNC
322  * request) or destroy a list of pages if an error occurred.
323  */
324 void
325 pvn_read_done(page_t *plist, int flags)
```



```
326 {
327         page_t *pp;

329         while (plist != NULL) {
330                 pp = plist;
331                 page_sub(&plist, pp);
332                 page_io_unlock(pp);
333                 if (flags & B_ERROR) {
334                         /*LINTED: constant in conditional context*/
335                         VN_DISPOSE(pp, B_INVAL, 0, kcred);
336                 } else {
337                         (void) page_release(pp, 0);
338                 }
339         }
340 }

342 /*
343  * Automagic pageout.
344  * When memory gets tight, start freeing pages popping out of the
345  * write queue.
346  */
347 int     write_free = 1;
348 pgcnt_t pages_before_pager = 200;       /* LMXXX */

350 /*
351  * Routine to be called when page-out's complete.
352  * The caller, typically VOP_PUTPAGE, has to explicity call this routine
353  * after waiting for i/o to complete (biowait) to free the list of
354  * pages associated with the buffer.  These pages must be locked
355  * before i/o is initiated.
356  *
357  * If a write error occurs, the pages are marked as modified
358  * so the write will be re-tried later.
359  */

361 void
362 pvn_write_done(page_t *plist, int flags)
363 {
364         int dfree = 0;
365         int pgrec = 0;
366         int pgout = 0;
367         int pgpgout = 0;
368         int anonpgout = 0;
369         int anonfree = 0;
370         int fspgout = 0;
371         int fsfree = 0;
372         int execpgout = 0;
373         int execfree = 0;
374         page_t *pp;
375         struct cpu *cpup;
376         struct vnode *vp = NULL;        /* for probe */
377         uint_t ppattr;
378         kmutex_t *vphm = NULL;

380         ASSERT((flags & B_READ) == 0);

382         /*
383          * If we are about to start paging anyway, start freeing pages.
384          */
385         if (write_free && freemem < lotsfree + pages_before_pager &&
386             (flags & B_ERROR) == 0) {
387                 flags |= B_FREE;
388         }

390         /*
391          * Handle each page involved in the i/o operation.
```

```
392          */
393         while (plist != NULL) {
394                 pp = plist;
395                 ASSERT(PAGE_LOCKED(pp) && page_iolock_assert(pp));
396                 page_sub(&plist, pp);
397
398                 /* Kernel probe support */
399                 if (vp == NULL)
400                         vp = pp->p_vnode;
401
402                 if (((flags & B_ERROR) == 0) && IS_VMODSORT(vp)) {
403                         /*
404                          * Move page to the top of the v_page list.
405                          * Skip pages modified during IO.
406                          */
407                         vphm = page_vnode_mutex(vp);
408                         mutex_enter(vphm);
409                         if ((pp->p_vpnext != pp) && !hat_ismod(pp)) {
410                                 page_vpsub(&vp->v_pages, pp);
411                                 page_vpadd(&vp->v_pages, pp);
412                         }
413                         mutex_exit(vphm);
414                 }
415
416                 if (flags & B_ERROR) {
417                         /*
418                          * Write operation failed.  We don't want
419                          * to destroy (or free) the page unless B_FORCE
420                          * is set. We set the mod bit again and release
421                          * all locks on the page so that it will get written
422                          * back again later when things are hopefully
423                          * better again.
424                          * If B_INVAL and B_FORCE is set we really have
425                          * to destroy the page.
426                          */
427                         if ((flags & (B_INVAL|B_FORCE)) == (B_INVAL|B_FORCE)) {
428                                 page_io_unlock(pp);
429                                 /*LINTED: constant in conditional context*/
430                                 VN_DISPOSE(pp, B_INVAL, 0, kcred);
431                         } else {
432                                 hat_setmod_only(pp);
433                                 page_io_unlock(pp);
434                                 page_unlock(pp);
435                         }
436                 } else if (flags & B_INVAL) {
437                         /*
438                          * XXX - Failed writes with B_INVAL set are
439                          * not handled appropriately.
440                          */
441                         page_io_unlock(pp);
442                         /*LINTED: constant in conditional context*/
443                         VN_DISPOSE(pp, B_INVAL, 0, kcred);
444                 } else if (flags & B_FREE ||!hat_page_is_mapped(pp)) {
445                         /*
446                          * Update statistics for pages being paged out
447                          */
448                         if (pp->p_vnode) {
449                                 if (IS_SWAPFSVP(pp->p_vnode)) {
450                                         anonpgout++;
451                                 } else {
452                                         if (pp->p_vnode->v_flag & VVMEXEC) {
453                                                 execpgout++;
454                                         } else {
455                                                 fspgout++;
456                                         }
457                                 }
```



```
458                         }
459                         page_io_unlock(pp);
460                         pgout = 1;
461                         pgpgout++;
462                         TRACE_1(TR_FAC_VM, TR_PAGE_WS_OUT,
463                             "page_ws_out:pp %p", pp);
464
465                         /*
466                          * The page_struct_lock need not be acquired to
467                          * examine "p_lckcnt" and "p_cowcnt" since we'll
468                          * have an "exclusive" lock if the upgrade succeeds.
469                          */
470                         if (page_tryupgrade(pp) &&
471                             pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
472                                 /*
473                                  * Check if someone has reclaimed the
474                                  * page.  If ref and mod are not set, no
475                                  * one is using it so we can free it.
476                                  * The rest of the system is careful
477                                  * to use the NOSYNC flag to unload
478                                  * translations set up for i/o w/o
479                                  * affecting ref and mod bits.
480                                  *
481                                  * Obtain a copy of the real hardware
482                                  * mod bit using hat_pagesync(pp, HAT_DONTZERO)
483                                  * to avoid having to flush the cache.
484                                  */
485                                 ppattr = hat_pagesync(pp, HAT_SYNC_DONTZERO |
486                                     HAT_SYNC_STOPON_MOD);
487                         ck_refmod:
488                                 if (!(ppattr & (P_REF | P_MOD))) {
489                                         if (hat_page_is_mapped(pp)) {
490                                                 /*
491                                                  * Doesn't look like the page
492                                                  * was modified so now we
493                                                  * really have to unload the
494                                                  * translations.  Meanwhile
495                                                  * another CPU could've
496                                                  * modified it so we have to
497                                                  * check again.  We don't loop
498                                                  * forever here because now
499                                                  * the translations are gone
500                                                  * and no one can get a new one
501                                                  * since we have the "exclusive"
502                                                  * lock on the page.
503                                                  */
504                                                 (void) hat_pageunload(pp,
505                                                     HAT_FORCE_PGUNLOAD);
506                                                 ppattr = hat_page_getattr(pp,
507                                                     P_REF | P_MOD);
508                                                 goto ck_refmod;
509                                         }
510                                         /*
511                                          * Update statistics for pages being
512                                          * freed
513                                          */
514                                         if (pp->p_vnode) {
515                                                 if (IS_SWAPFSVP(pp->p_vnode)) {
516                                                         anonfree++;
517                                                 } else {
518                                                         if (pp->p_vnode->v_flag
519                                                             & VVMEXEC) {
520                                                                 execfree++;
521                                                         } else {
522                                                                 fsfree++;
523                                                         }
```

```
524                                             }
525                                     }
526                                     /*LINTED: constant in conditional ctx*/
527                                     VN_DISPOSE(pp, B_FREE,
528                                         (flags & B_DONTNEED), kcred);
529                                     dfree++;
530                             } else {
531                                     page_unlock(pp);
532                                     pgrec++;
533                                     TRACE_1(TR_FAC_VM, TR_PAGE_WS_FREE,
534                                         "page_ws_free:pp %p", pp);
535                             }
536                     } else {
537                             /*
538                              * Page is either `locked' in memory
539                              * or was reclaimed and now has a
540                              * "shared" lock, so release it.
541                              */
542                             page_unlock(pp);
543                     }
544             } else {
545                     /*
546                      * Neither B_FREE nor B_INVAL nor B_ERROR.
547                      * Just release locks.
548                      */
549                     page_io_unlock(pp);
550                     page_unlock(pp);
551             }
552     }

554     CPU_STATS_ENTER_K();
555     cpup = CPU;             /* get cpup now that CPU cannot change */
556     CPU_STATS_ADDQ(cpup, vm, dfree, dfree);
557     CPU_STATS_ADDQ(cpup, vm, pgrec, pgrec);
558     CPU_STATS_ADDQ(cpup, vm, pgout, pgout);
559     CPU_STATS_ADDQ(cpup, vm, pgpgout, pgpgout);
560     CPU_STATS_ADDQ(cpup, vm, anonpgout, anonpgout);
561     CPU_STATS_ADDQ(cpup, vm, anonfree, anonfree);
562     CPU_STATS_ADDQ(cpup, vm, fspgout, fspgout);
563     CPU_STATS_ADDQ(cpup, vm, fsfree, fsfree);
564     CPU_STATS_ADDQ(cpup, vm, execpgout, execpgout);
565     CPU_STATS_ADDQ(cpup, vm, execfree, execfree);
566     CPU_STATS_EXIT_K();

568     /* Kernel probe */
569     TNF_PROBE_4(pageout, "vm pageio io", /* CSTYLED */,
570         tnf_opaque, vnode,                  vp,
571         tnf_ulong,  pages_pageout,          pgpgout,
572         tnf_ulong,  pages_freed,            dfree,
573         tnf_ulong,  pages_reclaimed,        pgrec);
574 }

576 /*
577  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED, B_DELWRI,
578  * B_TRUNC, B_FORCE}.  B_DELWRI indicates that this page is part of a kluster
579  * operation and is only to be considered if it doesn't involve any
580  * waiting here.  B_TRUNC indicates that the file is being truncated
581  * and so no i/o needs to be done. B_FORCE indicates that the page
582  * must be destroyed so don't try wrting it out.
583  *
584  * The caller must ensure that the page is locked.  Returns 1, if
585  * the page should be written back (the "iolock" is held in this
586  * case), or 0 if the page has been dealt with or has been
587  * unlocked.
588  */
589 int
```



```
590 pvn_getdirty(page_t *pp, int flags)
591 {
592         ASSERT((flags & (B_INVAL | B_FREE)) ?
593             PAGE_EXCL(pp) : PAGE_SHARED(pp));
594         ASSERT(PP_ISFREE(pp) == 0);

596         /*
597          * If trying to invalidate or free a logically `locked' page,
598          * forget it.  Don't need page_struct_lock to check p_lckcnt and
599          * p_cowcnt as the page is exclusively locked.
600          */
601         if ((flags & (B_INVAL | B_FREE)) && !(flags & (B_TRUNC|B_FORCE)) &&
602             (pp->p_lckcnt != 0 || pp->p_cowcnt != 0)) {
603                 page_unlock(pp);
604                 return (0);
605         }

607         /*
608          * Now acquire the i/o lock so we can add it to the dirty
609          * list (if necessary).  We avoid blocking on the i/o lock
610          * in the following cases:
611          *
612          *      If B_DELWRI is set, which implies that this request is
613          *      due to a klustering operartion.
614          *
615          *      If this is an async (B_ASYNC) operation and we are not doing
616          *      invalidation (B_INVAL) [The current i/o or fsflush will ensure
617          *      that the the page is written out].
618          */
619         if ((flags & B_DELWRI) || ((flags & (B_INVAL | B_ASYNC)) == B_ASYNC)) {
620                 if (!page_io_trylock(pp)) {
621                         page_unlock(pp);
622                         return (0);
623                 }
624         } else {
625                 page_io_lock(pp);
626         }

628         /*
629          * If we want to free or invalidate the page then
630          * we need to unload it so that anyone who wants
631          * it will have to take a minor fault to get it.
632          * Otherwise, we're just writing the page back so we
633          * need to sync up the hardwre and software mod bit to
634          * detect any future modifications.  We clear the
635          * software mod bit when we put the page on the dirty
636          * list.
637          */
638         if (flags & (B_INVAL | B_FREE)) {
639                 (void) hat_pageunload(pp, HAT_FORCE_PGUNLOAD);
640         } else {
641                 (void) hat_pagesync(pp, HAT_SYNC_ZERORM);
642         }

644         if (!hat_ismod(pp) || (flags & B_TRUNC)) {
645                 /*
646                  * Don't need to add it to the
647                  * list after all.
648                  */
649                 page_io_unlock(pp);
650                 if (flags & B_INVAL) {
651                         /*LINTED: constant in conditional context*/
652                         VN_DISPOSE(pp, B_INVAL, 0, kcred);
653                 } else if (flags & B_FREE) {
654                         /*LINTED: constant in conditional context*/
655                         VN_DISPOSE(pp, B_FREE, (flags & B_DONTNEED), kcred);
```

```
656                 } else {
657                         /*
658                          * This is advisory path for the callers
659                          * of VOP_PUTPAGE() who prefer freeing the
660                          * page _only_ if no one else is accessing it.
661                          * E.g. segmap_release()
662                          *
663                          * The above hat_ismod() check is useless because:
664                          * (1) we may not be holding SE_EXCL lock;
665                          * (2) we've not unloaded _all_ translations
666                          *
667                          * Let page_release() do the heavy-lifting.
668                          */
669                         (void) page_release(pp, 1);
670                 }
671                 return (0);
672         }

674         /*
675          * Page is dirty, get it ready for the write back
676          * and add page to the dirty list.
677          */
678         hat_clrrefmod(pp);

680         /*
681          * If we're going to free the page when we're done
682          * then we can let others try to use it starting now.
683          * We'll detect the fact that they used it when the
684          * i/o is done and avoid freeing the page.
685          */
686         if (flags & B_FREE)
687                 page_downgrade(pp);


690         TRACE_1(TR_FAC_VM, TR_PVN_GETDIRTY, "pvn_getdirty:pp %p", pp);

692         return (1);
693 }


696 /*ARGSUSED*/
697 static int
698 marker_constructor(void *buf, void *cdrarg, int kmflags)
699 {
700         page_t *mark = buf;
701         bzero(mark, sizeof (page_t));
702         mark->p_hash = PVN_VPLIST_HASH_TAG;
703         return (0);
704 }

706 void
707 pvn_init()
708 {
709         if (pvn_vmodsort_disable == 0)
710                 pvn_vmodsort_supported = hat_supported(HAT_VMODSORT, NULL);
711         marker_cache = kmem_cache_create("marker_cache",
712             sizeof (page_t), 0, marker_constructor,
713             NULL, NULL, NULL, NULL, 0);
714 }


717 /*
718  * Process a vnode's page list for all pages whose offset is >= off.
719  * Pages are to either be free'd, invalidated, or written back to disk.
720  *
721  * An "exclusive" lock is acquired for each page if B_INVAL or B_FREE
```



```
722  * is specified, otherwise they are "shared" locked.
723  *
724  * Flags are {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED, B_TRUNC}
725  *
726  * Special marker page_t's are inserted in the list in order
727  * to keep track of where we are in the list when locks are dropped.
728  *
729  * Note the list is circular and insertions can happen only at the
730  * head and tail of the list. The algorithm ensures visiting all pages
731  * on the list in the following way:
732  *
733  *    Drop two marker pages at the end of the list.
734  *
735  *    Move one marker page backwards towards the start of the list until
736  *    it is at the list head, processing the pages passed along the way.
737  *
738  *    Due to race conditions when the vphm mutex is dropped, additional pages
739  *    can be added to either end of the list, so we'll continue to move
740  *    the marker and process pages until it is up against the end marker.
741  *
742  * There is one special exit condition. If we are processing a VMODSORT
743  * vnode and only writing back modified pages, we can stop as soon as
744  * we run into an unmodified page.  This makes fsync(3) operations fast.
745  */
746 int
747 pvn_vplist_dirty(
748         vnode_t         *vp,
749         u_offset_t      off,
750         int             (*putapage)(vnode_t *, page_t *, u_offset_t *,
751                         size_t *, int, cred_t *),
752         int             flags,
753         cred_t          *cred)
754 {
755         page_t          *pp;
756         page_t          *mark;          /* marker page that moves toward head */
757         page_t          *end;           /* marker page at end of list */
758         int             err = 0;
759         int             error;
760         kmutex_t        *vphm;
761         se_t            se;
762         page_t          **where_to_move;

764         ASSERT(vp->v_type != VCHR);

766         if (vp->v_pages == NULL)
767                 return (0);


770         /*
771          * Serialize vplist_dirty operations on this vnode by setting VVMLOCK.
772          *
773          * Don't block on VVMLOCK if B_ASYNC is set. This prevents sync()
774          * from getting blocked while flushing pages to a dead NFS server.
775          */
776         mutex_enter(&vp->v_lock);
777         if ((vp->v_flag & VVMLOCK) && (flags & B_ASYNC)) {
778                 mutex_exit(&vp->v_lock);
779                 return (EAGAIN);
780         }

782         while (vp->v_flag & VVMLOCK)
783                 cv_wait(&vp->v_cv, &vp->v_lock);

785         if (vp->v_pages == NULL) {
786                 mutex_exit(&vp->v_lock);
787                 return (0);
```

```
788         }
789
790         vp->v_flag |= VVMLOCK;
791         mutex_exit(&vp->v_lock);
792
793
794         /*
795          * Set up the marker pages used to walk the list
796          */
797         end = kmem_cache_alloc(marker_cache, KM_SLEEP);
798         end->p_vnode = vp;
799         end->p_offset = (u_offset_t)-2;
800         mark = kmem_cache_alloc(marker_cache, KM_SLEEP);
801         mark->p_vnode = vp;
802         mark->p_offset = (u_offset_t)-1;
803
804         /*
805          * Grab the lock protecting the vnode's page list
806          * note that this lock is dropped at times in the loop.
807          */
808         vphm = page_vnode_mutex(vp);
809         mutex_enter(vphm);
810         if (vp->v_pages == NULL)
811                 goto leave;
812
813         /*
814          * insert the markers and loop through the list of pages
815          */
816         page_vpadd(&vp->v_pages->p_vpprev->p_vpnext, mark);
817         page_vpadd(&mark->p_vpnext, end);
818         for (;;) {
819
820                 /*
821                  * If only doing an async write back, then we can
822                  * stop as soon as we get to start of the list.
823                  */
824                 if (flags == B_ASYNC && vp->v_pages == mark)
825                         break;
826
827                 /*
828                  * otherwise stop when we've gone through all the pages
829                  */
830                 if (mark->p_vpprev == end)
831                         break;
832
833                 pp = mark->p_vpprev;
834                 if (vp->v_pages == pp)
835                         where_to_move = &vp->v_pages;
836                 else
837                         where_to_move = &pp->p_vpprev->p_vpnext;
838
839                 ASSERT(pp->p_vnode == vp);
840
841                 /*
842                  * If just flushing dirty pages to disk and this vnode
843                  * is using a sorted list of pages, we can stop processing
844                  * as soon as we find an unmodified page. Since all the
845                  * modified pages are visited first.
846                  */
847                 if (IS_VMODSORT(vp) &&
848                     !(flags & (B_INVAL | B_FREE | B_TRUNC))) {
849                         if (!hat_ismod(pp) && !page_io_locked(pp)) {
850 #ifdef  DEBUG
851                                 /*
852                                  * For debug kernels examine what should be
853                                  * all the remaining clean pages, asserting
```



```
854                                  * that they are not modified.
855                                  */
856                                 page_t  *chk = pp;
857                                 int     attr;
858
859                                 page_vpsub(&vp->v_pages, mark);
860                                 page_vpadd(where_to_move, mark);
861                                 do {
862                                         chk = chk->p_vpprev;
863                                         ASSERT(chk != end);
864                                         if (chk == mark)
865                                                 continue;
866                                         attr = hat_page_getattr(chk, P_MOD |
867                                             P_REF);
868                                         if ((attr & P_MOD) == 0)
869                                                 continue;
870                                         panic("v_pages list not all clean: "
871                                             "page_t*=%p vnode=%p off=%lx "
872                                             "attr=0x%x last clean page_t*=%p\n",
873                                             (void *)chk, (void *)chk->p_vnode,
874                                             (long)chk->p_offset, attr,
875                                             (void *)pp);
876                                 } while (chk != vp->v_pages);
877 #endif
878                                 break;
879                         } else if (!(flags & B_ASYNC) && !hat_ismod(pp)) {
880                                 /*
881                                  * Couldn't get io lock, wait until IO is done.
882                                  * Block only for sync IO since we don't want
883                                  * to block async IO.
884                                  */
885                                 mutex_exit(vphm);
886                                 page_io_wait(pp);
887                                 mutex_enter(vphm);
888                                 continue;
889                         }
890                 }
891
892                 /*
893                  * Skip this page if the offset is out of the desired range.
894                  * Just move the marker and continue.
895                  */
896                 if (pp->p_offset < off) {
897                         page_vpsub(&vp->v_pages, mark);
898                         page_vpadd(where_to_move, mark);
899                         continue;
900                 }
901
902                 /*
903                  * If we are supposed to invalidate or free this
904                  * page, then we need an exclusive lock.
905                  */
906                 se = (flags & (B_INVAL | B_FREE)) ? SE_EXCL : SE_SHARED;
907
908                 /*
909                  * We must acquire the page lock for all synchronous
910                  * operations (invalidate, free and write).
911                  */
912                 if ((flags & B_INVAL) != 0 || (flags & B_ASYNC) == 0) {
913                         /*
914                          * If the page_lock() drops the mutex
915                          * we must retry the loop.
916                          */
917                         if (!page_lock(pp, se, vphm, P_NO_RECLAIM))
918                                 continue;
```

```
920                         /*
921                          * It's ok to move the marker page now.
922                          */
923                         page_vpsub(&vp->v_pages, mark);
924                         page_vpadd(where_to_move, mark);
925                 } else {

927                         /*
928                          * update the marker page for all remaining cases
929                          */
930                         page_vpsub(&vp->v_pages, mark);
931                         page_vpadd(where_to_move, mark);

933                         /*
934                          * For write backs, If we can't lock the page, it's
935                          * invalid or in the process of being destroyed.  Skip
936                          * it, assuming someone else is writing it.
937                          */
938                         if (!page_trylock(pp, se))
939                                 continue;
940                 }

942                 ASSERT(pp->p_vnode == vp);

944                 /*
945                  * Successfully locked the page, now figure out what to
946                  * do with it. Free pages are easily dealt with, invalidate
947                  * if desired or just go on to the next page.
948                  */
949                 if (PP_ISFREE(pp)) {
950                         if ((flags & B_INVAL) == 0) {
951                                 page_unlock(pp);
952                                 continue;
953                         }

955                         /*
956                          * Invalidate (destroy) the page.
957                          */
958                         mutex_exit(vphm);
959                         page_destroy_free(pp);
960                         mutex_enter(vphm);
961                         continue;
962                 }

964                 /*
965                  * pvn_getdirty() figures out what do do with a dirty page.
966                  * If the page is dirty, the putapage() routine will write it
967                  * and will kluster any other adjacent dirty pages it can.
968                  *
969                  * pvn_getdirty() and '(*putapage)' unlock the page.
970                  */
971                 mutex_exit(vphm);
972                 if (pvn_getdirty(pp, flags)) {
973                         error = (*putapage)(vp, pp, NULL, NULL, flags, cred);
974                         if (!err)
975                                 err = error;
976                 }
977                 mutex_enter(vphm);
978         }
979         page_vpsub(&vp->v_pages, mark);
980         page_vpsub(&vp->v_pages, end);

982 leave:
983         /*
984          * Release v_pages mutex, also VVMLOCK and wakeup blocked thrds
985          */
```



```
986         mutex_exit(vphm);
987         kmem_cache_free(marker_cache, mark);
988         kmem_cache_free(marker_cache, end);
989         mutex_enter(&vp->v_lock);
990         vp->v_flag &= ~VVMLOCK;
991         cv_broadcast(&vp->v_cv);
992         mutex_exit(&vp->v_lock);
993         return (err);
994 }

996 /*
997  * Walk the vp->v_pages list, for every page call the callback function
998  * pointed by *page_check. If page_check returns non-zero, then mark the
999  * page as modified and if VMODSORT is set, move it to the end of v_pages
1000 * list. Moving makes sense only if we have at least two pages - this also
1001 * avoids having v_pages temporarily being NULL after calling page_vpsub()
1002 * if there was just one page.
1003 */
1004 void
1005 pvn_vplist_setdirty(vnode_t *vp, int (*page_check)(page_t *))
1006 {
1007         page_t  *pp, *next, *end;
1008         kmutex_t        *vphm;
1009         int     shuffle;

1011         vphm = page_vnode_mutex(vp);
1012         mutex_enter(vphm);

1014         if (vp->v_pages == NULL) {
1015                 mutex_exit(vphm);
1016                 return;
1017         }

1019         end = vp->v_pages->p_vpprev;
1020         shuffle = IS_VMODSORT(vp) && (vp->v_pages != end);
1021         pp = vp->v_pages;

1023         for (;;) {
1024                 next = pp->p_vpnext;
1025                 if (pp->p_hash != PVN_VPLIST_HASH_TAG && page_check(pp)) {
1026                         /*
1027                          * hat_setmod_only() in contrast to hat_setmod() does
1028                          * not shuffle the pages and does not grab the mutex
1029                          * page_vnode_mutex. Exactly what we need.
1030                          */
1031                         hat_setmod_only(pp);
1032                         if (shuffle) {
1033                                 page_vpsub(&vp->v_pages, pp);
1034                                 ASSERT(vp->v_pages != NULL);
1035                                 page_vpadd(&vp->v_pages->p_vpprev->p_vpnext,
1036                                     pp);
1037                         }
1038                 }
1039                 /* Stop if we have just processed the last page. */
1040                 if (pp == end)
1041                         break;
1042                 pp = next;
1043         }

1045         mutex_exit(vphm);
1046 }

1048 /*
1049  * Zero out zbytes worth of data. Caller should be aware that this
1050  * routine may enter back into the fs layer (xxx_getpage). Locks
1051  * that the xxx_getpage routine may need should not be held while
```

```
1052  * calling this.
1053  */
1054 void
1055 pvn_vpzero(struct vnode *vp, u_offset_t vplen, size_t zbytes)
1056 {
1057         caddr_t addr;

1059         ASSERT(vp->v_type != VCHR);

1061         if (vp->v_pages == NULL)
1062                 return;

1064         /*
1065          * zbytes may be zero but there still may be some portion of
1066          * a page which needs clearing (since zbytes is a function
1067          * of filesystem block size, not pagesize.)
1068          */
1069         if (zbytes == 0 && (PAGESIZE - (vplen & PAGEOFFSET)) == 0)
1070                 return;

1072         /*
1073          * We get the last page and handle the partial
1074          * zeroing via kernel mappings.  This will make the page
1075          * dirty so that we know that when this page is written
1076          * back, the zeroed information will go out with it.  If
1077          * the page is not currently in memory, then the kzero
1078          * operation will cause it to be brought it.  We use kzero
1079          * instead of bzero so that if the page cannot be read in
1080          * for any reason, the system will not panic.  We need
1081          * to zero out a minimum of the fs given zbytes, but we
1082          * might also have to do more to get the entire last page.
1083          */

1085         if ((zbytes + (vplen & MAXBOFFSET)) > MAXBSIZE)
1086                 panic("pvn_vptrunc zbytes");
1087         addr = segmap_getmapflt(segkmap, vp, vplen,
1088             MAX(zbytes, PAGESIZE - (vplen & PAGEOFFSET)), 1, S_WRITE);
1089         (void) kzero(addr + (vplen & MAXBOFFSET),
1090             MAX(zbytes, PAGESIZE - (vplen & PAGEOFFSET)));
1091         (void) segmap_release(segkmap, addr, SM_WRITE | SM_ASYNC);
1092 }

1094 /*
1095  * Handles common work of the VOP_GETPAGE routines by iterating page by page
1096  * calling the getpage helper for each.
  23  * Handles common work of the VOP_GETPAGE routines when more than
  24  * one page must be returned by calling a file system specific operation
  25  * to do most of the work.  Must be called with the vp already locked
  26  * by the VOP_GETPAGE routine.
1097  */
1098 int
1099 pvn_getpages(
1100         int (*getpage)(vnode_t *, u_offset_t, size_t, uint_t *, page_t *[],
1101                 size_t, struct seg *, caddr_t, enum seg_rw, cred_t *),
1102         struct vnode *vp,
1103         u_offset_t off,
1104         size_t len,
1105         uint_t *protp,
1106         page_t *pl[],
1107         size_t plsz,
1108         struct seg *seg,
1109         caddr_t addr,
1110         enum seg_rw rw,
1111         struct cred *cred)
1112 {
1113         page_t **ppp;
```



```
1114         u_offset_t o, eoff;
1115         size_t sz, xlen;
1116         int err;

1118         /* ensure that we have enough space */
1119         ASSERT(pl == NULL || plsz >= len);
  48         ASSERT(plsz >= len);            /* insure that we have enough space */

1121         /*
1122          * Loop one page at a time and let getapage function fill
1123          * in the next page in array.  We only allow one page to be
1124          * returned at a time (except for the last page) so that we
1125          * don't have any problems with duplicates and other such
1126          * painful problems.  This is a very simple minded algorithm,
1127          * but it does the job correctly.  We hope that the cost of a
1128          * getapage call for a resident page that we might have been
1129          * able to get from an earlier call doesn't cost too much.
1130          */
1131         ppp = pl;
1132         sz = (pl != NULL) ? PAGESIZE : 0;
  61         sz = PAGESIZE;
1133         eoff = off + len;
1134         xlen = len;
1135         for (o = off; o < eoff; o += PAGESIZE, addr += PAGESIZE,
1136             xlen -= PAGESIZE) {
1137                 if (o + PAGESIZE >= eoff && pl != NULL) {
  66                 if (o + PAGESIZE >= eoff) {
1138                         /*
1139                          * Last time through - allow the all of
1140                          * what's left of the pl[] array to be used.
1141                          */
1142                         sz = plsz - (o - off);
1143                 }
1144                 err = (*getpage)(vp, o, xlen, protp, ppp, sz, seg, addr,
1145                     rw, cred);
1146                 if (err) {
1147                         /*
1148                          * Release any pages we already got.
1149                          */
1150                         if (o > off && pl != NULL) {
1151                                 for (ppp = pl; *ppp != NULL; *ppp++ = NULL)
1152                                         (void) page_release(*ppp, 1);
1153                         }
1154                         break;
1155                 }
1156                 if (pl != NULL)
1157                         ppp++;
1158         }
1159         return (err);
1160 }
```

_____***unchanged_portion_omitted_***