****

49002 Fri Feb 27 16:32:49 2015
new/usr/src/uts/common/vm/page.h
vm: be careful about empty statements

****

______unchanged_portion_omitted_

```
/*
 * Callers of page_try_reclaim_lock and page_lock_es can use this flag
 * to get SE_EXCL access before reader/writers are given access.
 */
#define SE_EXCL_WANTED  0x02

/*
 * All page_*lock() requests will be denied unless this flag is set in
 * the 'es' parameter.
 */
#define SE_RETIRED      0x04

#endif  /* _KERNEL | _KMEMUSER */

typedef int     selock_t;

/*
 * Define VM_STATS to turn on all sorts of statistic gathering about
 * the VM layer.  By default, it is only turned on when DEBUG is
 * also defined.
 */
#ifdef DEBUG
#define VM_STATS
#endif  /* DEBUG */

#ifdef VM_STATS
#define VM_STAT_ADD(stat)                       (stat)++
#define VM_STAT_COND_ADD(cond, stat)            ((void) (!(cond) || (stat)++))
#else
#define VM_STAT_ADD(stat)                       do { } while (0)
#define VM_STAT_COND_ADD(cond, stat)            do { } while (0)
#define VM_STAT_ADD(stat)
#define VM_STAT_COND_ADD(cond, stat)
#endif  /* VM_STATS */

#ifdef _KERNEL

/*
 * PAGE_LLOCK_SIZE is 2 * NCPU, but no smaller than 128.
 * PAGE_LLOCK_SHIFT is log2(PAGE_LLOCK_SIZE).
 *
 * We use ? : instead of #if because <vm/page.h> is included everywhere;
 * NCPU_P2 is only a constant in the "unix" module.
 *
 */
#define PAGE_LLOCK_SHIFT \
            ((unsigned)(((2*NCPU_P2) > 128) ? NCPU_LOG2 + 1 : 7))

#define PAGE_LLOCK_SIZE (1ul << PAGE_LLOCK_SHIFT)

/*
 * The number of low order 0 (or less variable) bits in the page_t address.
 */
#if defined(__sparc)
#define PP_SHIFT                7
#else
#define PP_SHIFT                6
#endif
```



```
/*
 * pp may be the root of a large page, and many low order bits will be 0.
 * Shift and XOR multiple times to capture the good bits across the range of
 * possible page sizes.
 */
#define PAGE_LLOCK_HASH(pp)     \
        (((((uintptr_t)(pp) >> PP_SHIFT) ^ \
        ((uintptr_t)(pp) >> (PAGE_LLOCK_SHIFT + PP_SHIFT))) ^ \
        ((uintptr_t)(pp) >> ((PAGE_LLOCK_SHIFT * 2) + PP_SHIFT)) ^ \
        ((uintptr_t)(pp) >> ((PAGE_LLOCK_SHIFT * 3) + PP_SHIFT))) & \
        (PAGE_LLOCK_SIZE - 1))

#define page_struct_lock(pp)    \
        mutex_enter(&page_llocks[PAGE_LLOCK_HASH(PP_PAGEROOT(pp))].pad_mutex)
#define page_struct_unlock(pp)  \
        mutex_exit(&page_llocks[PAGE_LLOCK_HASH(PP_PAGEROOT(pp))].pad_mutex)

#endif  /* _KERNEL */

#include <sys/t_lock.h>

struct as;

/*
 * Each physical page has a page structure, which is used to maintain
 * these pages as a cache.  A page can be found via a hashed lookup
 * based on the [vp, offset].  If a page has an [vp, offset] identity,
 * then it is entered on a doubly linked circular list off the
 * vnode using the vpnext/vpprev pointers.   If the p_free bit
 * is on, then the page is also on a doubly linked circular free
 * list using next/prev pointers.  If the "p_selock" and "p_iolock"
 * are held, then the page is currently being read in (exclusive p_selock)
 * or written back (shared p_selock).  In this case, the next/prev pointers
 * are used to link the pages together for a consecutive i/o request.  If
 * the page is being brought in from its backing store, then other processes
 * will wait for the i/o to complete before attaching to the page since it
 * will have an "exclusive" lock.
 *
 * Each page structure has the locks described below along with
 * the fields they protect:
 *
 *      p_selock        This is a per-page shared/exclusive lock that is
 *                      used to implement the logical shared/exclusive
 *                      lock for each page.  The "shared" lock is normally
 *                      used in most cases while the "exclusive" lock is
 *                      required to destroy or retain exclusive access to
 *                      a page (e.g., while reading in pages).  The appropriate
 *                      lock is always held whenever there is any reference
 *                      to a page structure (e.g., during i/o).
 *                      (Note that with the addition of the "writer-lock-wanted"
 *                      semantics (via SE_EWANTED), threads must not acquire
 *                      multiple reader locks or else a deadly embrace will
 *                      occur in the following situation: thread 1 obtains a
 *                      reader lock; next thread 2 fails to get a writer lock
 *                      but specified SE_EWANTED so it will wait by either
 *                      blocking (when using page_lock_es) or spinning while
 *                      retrying (when using page_try_reclaim_lock) until the
 *                      reader lock is released; then thread 1 attempts to
 *                      get another reader lock but is denied due to
 *                      SE_EWANTED being set, and now both threads are in a
 *                      deadly embrace.)
 *
 *                              p_hash
 *                              p_vnode
 *                              p_offset
 *
```

```
 *                              p_free
 *                              p_age
 *
 *      p_iolock        This is a binary semaphore lock that provides
 *                      exclusive access to the i/o list links in each
 *                      page structure.  It is always held while the page
 *                      is on an i/o list (i.e., involved in i/o).  That is,
 *                      even though a page may be only 'shared' locked
 *                      while it is doing a write, the following fields may
 *                      change anyway.  Normally, the page must be
 *                      'exclusively' locked to change anything in it.
 *
 *                              p_next
 *                              p_prev
 *
 * The following fields are protected by the global page_llocks[]:
 *
 *                              p_lckcnt
 *                              p_cowcnt
 *
 * The following lists are protected by the global page_freelock:
 *
 *                              page_cachelist
 *                              page_freelist
 *
 * The following, for our purposes, are protected by
 * the global freemem_lock:
 *
 *                              freemem
 *                              freemem_wait
 *                              freemem_cv
 *
 * The following fields are protected by hat layer lock(s).  When a page
 * structure is not mapped and is not associated with a vnode (after a call
 * to page_hashout() for example) the p_nrm field may be modified with out
 * holding the hat layer lock:
 *
 *                              p_nrm
 *                              p_mapping
 *                              p_share
 *
 * The following field is file system dependent.  How it is used and
 * the locking strategies applied are up to the individual file system
 * implementation.
 *
 *                              p_fsdata
 *
 * The page structure is used to represent and control the system's
 * physical pages.  There is one instance of the structure for each
 * page that is not permenately allocated.  For example, the pages that
 * hold the page structures are permanently held by the kernel
 * and hence do not need page structures to track them.  The array
 * of page structures is allocated early on in the kernel's life and
 * is based on the amount of available physical memory.
 *
 * Each page structure may simultaneously appear on several linked lists.
 * The lists are:  hash list, free or in i/o list, and a vnode's page list.
 * Each type of list is protected by a different group of mutexes as described
 * below:
 *
 * The hash list is used to quickly find a page when the page's vnode and
 * offset within the vnode are known.  Each page that is hashed is
 * connected via the 'p_hash' field.  The anchor for each hash is in the
 * array 'page_hash'.  An array of mutexes, 'ph_mutex', protects the
 * lists anchored by page_hash[].  To either search or modify a given hash
 * list, the appropriate mutex in the ph_mutex array must be held.
```



```
 *
 * The free list contains pages that are 'free to be given away'.  For
 * efficiency reasons, pages on this list are placed in two catagories:
 * pages that are still associated with a vnode, and pages that are not
 * associated with a vnode.  Free pages always have their 'p_free' bit set,
 * free pages that are still associated with a vnode also have their
 * 'p_age' bit set.  Pages on the free list are connected via their
 * 'p_next' and 'p_prev' fields.  When a page is involved in some sort
 * of i/o, it is not free and these fields may be used to link associated
 * pages together.  At the moment, the free list is protected by a
 * single mutex 'page_freelock'.  The list of free pages still associated
 * with a vnode is anchored by 'page_cachelist' while other free pages
 * are anchored in architecture dependent ways (to handle page coloring etc.).
 *
 * Pages associated with a given vnode appear on a list anchored in the
 * vnode by the 'v_pages' field.  They are linked together with
 * 'p_vpnext' and 'p_vpprev'.  The field 'p_offset' contains a page's
 * offset within the vnode.  The pages on this list are not kept in
 * offset order.  These lists, in a manner similar to the hash lists,
 * are protected by an array of mutexes called 'vph_hash'.  Before
 * searching or modifying this chain the appropriate mutex in the
 * vph_hash[] array must be held.
 *
 * Again, each of the lists that a page can appear on is protected by a
 * mutex.  Before reading or writing any of the fields comprising the
 * list, the appropriate lock must be held.  These list locks should only
 * be held for very short intervals.
 *
 * In addition to the list locks, each page structure contains a
 * shared/exclusive lock that protects various fields within it.
 * To modify one of these fields, the 'p_selock' must be exclusively held.
 * To read a field with a degree of certainty, the lock must be at least
 * held shared.
 *
 * Removing a page structure from one of the lists requires holding
 * the appropriate list lock and the page's p_selock.  A page may be
 * prevented from changing identity, being freed, or otherwise modified
 * by acquiring p_selock shared.
 *
 * To avoid deadlocks, a strict locking protocol must be followed.  Basically
 * there are two cases:  In the first case, the page structure in question
 * is known ahead of time (e.g., when the page is to be added or removed
 * from a list).  In the second case, the page structure is not known and
 * must be found by searching one of the lists.
 *
 * When adding or removing a known page to one of the lists, first the
 * page must be exclusively locked (since at least one of its fields
 * will be modified), second the lock protecting the list must be acquired,
 * third the page inserted or deleted, and finally the list lock dropped.
 *
 * The more interesting case occures when the particular page structure
 * is not known ahead of time.  For example, when a call is made to
 * page_lookup(), it is not known if a page with the desired (vnode and
 * offset pair) identity exists.  So the appropriate mutex in ph_mutex is
 * acquired, the hash list searched, and if the desired page is found
 * an attempt is made to lock it.  The attempt to acquire p_selock must
 * not block while the hash list lock is held.  A deadlock could occure
 * if some other process was trying to remove the page from the list.
 * The removing process (following the above protocol) would have exclusively
 * locked the page, and be spinning waiting to acquire the lock protecting
 * the hash list.  Since the searching process holds the hash list lock
 * and is waiting to acquire the page lock, a deadlock occurs.
 *
 * The proper scheme to follow is: first, lock the appropriate list,
 * search the list, and if the desired page is found either use
 * page_trylock() (which will not block) or pass the address of the
```

```
 * list lock to page_lock().  If page_lock() can not acquire the page's
 * lock, it will drop the list lock before going to sleep.  page_lock()
 * returns a value to indicate if the list lock was dropped allowing the
 * calling program to react appropriately (i.e., retry the operation).
 *
 * If the list lock was dropped before the attempt at locking the page
 * was made, checks would have to be made to ensure that the page had
 * not changed identity before its lock was obtained.  This is because
 * the interval between dropping the list lock and acquiring the page
 * lock is indeterminate.
 *
 * In addition, when both a hash list lock (ph_mutex[]) and a vnode list
 * lock (vph_mutex[]) are needed, the hash list lock must be acquired first.
 * The routine page_hashin() is a good example of this sequence.
 * This sequence is ASSERTed by checking that the vph_mutex[] is not held
 * just before each acquisition of one of the mutexs in ph_mutex[].
 *
 * So, as a quick summary:
 *
 *      pse_mutex[]'s protect the p_selock and p_cv fields.
 *
 *      p_selock protects the p_free, p_age, p_vnode, p_offset and p_hash,
 *
 *      ph_mutex[]'s protect the page_hash[] array and its chains.
 *
 *      vph_mutex[]'s protect the v_pages field and the vp page chains.
 *
 *      First lock the page, then the hash chain, then the vnode chain.  When
 *      this is not possible `trylocks' must be used.  Sleeping while holding
 *      any of these mutexes (p_selock is not a mutex) is not allowed.
 *
 *
 *      field           reading         writing             ordering
 *      ===================================================================
 *      p_vnode         p_selock(E,S)   p_selock(E)
 *      p_offset
 *      p_free
 *      p_age
 *      =================================================================
 *      p_hash          p_selock(E,S)   p_selock(E) &&      p_selock, ph_mutex
 *                                      ph_mutex[]
 *      =================================================================
 *      p_vpnext        p_selock(E,S)   p_selock(E) &&      p_selock, vph_mutex
 *      p_vpprev                        vph_mutex[]
 *      =================================================================
 *      When the p_free bit is set:
 *
 *      p_next          p_selock(E,S)   p_selock(E) &&      p_selock,
 *      p_prev                          page_freelock       page_freelock
 *
 *      When the p_free bit is not set:
 *
 *      p_next          p_selock(E,S)   p_selock(E) &&      p_selock, p_iolock
 *      p_prev                          p_iolock
 *      =================================================================
 *      p_selock        pse_mutex[]     pse_mutex[]         can't acquire any
 *      p_cv                                                other mutexes or
 *                                                          sleep while holding
 *                                                          this lock.
 *      =================================================================
 *      p_lckcnt        p_selock(E,S)   p_selock(E)
 *                                          OR
 *                                      p_selock(S) &&
 *                                      page_llocks[]
 *      p_cowcnt
 *      =================================================================
```



```
 *      p_nrm           hat layer lock  hat layer lock
 *      p_mapping
 *      p_pagenum
 *      =================================================================
 *
 *      where:
 *              E----> exclusive version of p_selock.
 *              S----> shared version of p_selock.
 *
 *
 *      Global data structures and variable:
 *
 *      field           reading         writing             ordering
 *      =================================================================
 *      page_hash[]     ph_mutex[]      ph_mutex[]          can hold this lock
 *                                                          before acquiring
 *                                                          a vph_mutex or
 *                                                          pse_mutex.
 *      =================================================================
 *      vp->v_pages     vph_mutex[]     vph_mutex[]         can only acquire
 *                                                          a pse_mutex while
 *                                                          holding this lock.
 *      =================================================================
 *      page_cachelist  page_freelock   page_freelock       can't acquire any
 *      page_freelist   page_freelock   page_freelock
 *      =================================================================
 *      freemem         freemem_lock    freemem_lock        can't acquire any
 *      freemem_wait                                        other mutexes while
 *      freemem_cv                                          holding this mutex.
 *      =================================================================
 *
 * Page relocation, PG_NORELOC and P_NORELOC.
 *
 * Pages may be relocated using the page_relocate() interface. Relocation
 * involves moving the contents and identity of a page to another, free page.
 * To relocate a page, the SE_EXCL lock must be obtained. The way to prevent
 * a page from being relocated is to hold the SE_SHARED lock (the SE_EXCL
 * lock must not be held indefinitely). If the page is going to be held
 * SE_SHARED indefinitely, then the PG_NORELOC hint should be passed
 * to page_create_va so that pages that are prevented from being relocated
 * can be managed differently by the platform specific layer.
 *
 * Pages locked in memory using page_pp_lock (p_lckcnt/p_cowcnt != 0)
 * are guaranteed to be held in memory, but can still be relocated
 * providing the SE_EXCL lock can be obtained.
 *
 * The P_NORELOC bit in the page_t.p_state field is provided for use by
 * the platform specific code in managing pages when the PG_NORELOC
 * hint is used.
 *
 * Memory delete and page locking.
 *
 * The set of all usable pages is managed using the global page list as
 * implemented by the memseg structure defined below. When memory is added
 * or deleted this list changes. Additions to this list guarantee that the
 * list is never corrupt.  In order to avoid the necessity of an additional
 * lock to protect against failed accesses to the memseg being deleted and,
 * more importantly, the page_ts, the memseg structure is never freed and the
 * page_t virtual address space is remapped to a page (or pages) of
 * zeros.  If a page_t is manipulated while it is p_selock'd, or if it is
 * locked indirectly via a hash or freelist lock, it is not possible for
 * memory delete to collect the page and so that part of the page list is
 * prevented from being deleted. If the page is referenced outside of one
 * of these locks, it is possible for the page_t being referenced to be
 * deleted.  Examples of this are page_t pointers returned by
 * page_numtopp_nolock, page_first and page_next.  Providing the page_t
```

```
 * is re-checked after taking the p_selock (for p_vnode != NULL), the
 * remapping to the zero pages will be detected.
 *
 *
 * Page size (p_szc field) and page locking.
 *
 * p_szc field of free pages is changed by free list manager under freelist
 * locks and is of no concern to the rest of VM subsystem.
 *
 * p_szc changes of allocated anonymous (swapfs) can only be done only after
 * exclusively locking all constituent pages and calling hat_pageunload() on
 * each of them. To prevent p_szc changes of non free anonymous (swapfs) large
 * pages it's enough to either lock SHARED any of constituent pages or prevent
 * hat_pageunload() by holding hat level lock that protects mapping lists (this
 * method is for hat code only)
 *
 * To increase (promote) p_szc of allocated non anonymous file system pages
 * one has to first lock exclusively all involved constituent pages and call
 * hat_pageunload() on each of them. To prevent p_szc promote it's enough to
 * either lock SHARED any of constituent pages that will be needed to make a
 * large page or prevent hat_pageunload() by holding hat level lock that
 * protects mapping lists (this method is for hat code only).
 *
 * To decrease (demote) p_szc of an allocated non anonymous file system large
 * page one can either use the same method as used for changeing p_szc of
 * anonymous large pages or if it's not possible to lock all constituent pages
 * exclusively a different method can be used. In the second method one only
 * has to exclusively lock one of constituent pages but then one has to
 * acquire further locks by calling page_szc_lock() and
 * hat_page_demote(). hat_page_demote() acquires hat level locks and then
 * demotes the page. This mechanism relies on the fact that any code that
 * needs to prevent p_szc of a file system large page from changeing either
 * locks all constituent large pages at least SHARED or locks some pages at
 * least SHARED and calls page_szc_lock() or uses hat level page locks.
 * Demotion using this method is implemented by page_demote_vp_pages().
 * Please see comments in front of page_demote_vp_pages(), hat_page_demote()
 * and page_szc_lock() for more details.
 *
 * Lock order: p_selock, page_szc_lock, ph_mutex/vph_mutex/freelist,
 * hat level locks.
 */

typedef struct page {
        u_offset_t      p_offset;       /* offset into vnode for this page */
        struct vnode    *p_vnode;       /* vnode that this page is named by */
        selock_t        p_selock;       /* shared/exclusive lock on the page */
#if defined(_LP64)
        uint_t          p_vpmref;       /* vpm ref - index of the vpmap_t */
#endif
        struct page     *p_hash;        /* hash by [vnode, offset] */
        struct page     *p_vpnext;      /* next page in vnode list */
        struct page     *p_vpprev;      /* prev page in vnode list */
        struct page     *p_next;        /* next page in free/intrans lists */
        struct page     *p_prev;        /* prev page in free/intrans lists */
        ushort_t        p_lckcnt;       /* number of locks on page data */
        ushort_t        p_cowcnt;       /* number of copy on write lock */
        kcondvar_t      p_cv;           /* page struct's condition var */
        kcondvar_t      p_io_cv;        /* for iolock */
        uchar_t         p_iolock_state; /* replaces p_iolock */
        volatile uchar_t p_szc;         /* page size code */
        uchar_t         p_fsdata;       /* file system dependent byte */
        uchar_t         p_state;        /* p_free, p_noreloc */
        uchar_t         p_nrm;          /* non-cache, ref, mod readonly bits */
#if defined(__sparc)
        uchar_t         p_vcolor;       /* virtual color */
#else
```



```
        uchar_t         p_embed;        /* x86 - changes p_mapping & p_index */
#endif
        uchar_t         p_index;        /* MPSS mapping info. Not used on x86 */
        uchar_t         p_toxic;        /* page has an unrecoverable error */
        void            *p_mapping;     /* hat specific translation info */
        pfn_t           p_pagenum;      /* physical page number */

        uint_t          p_share;        /* number of translations */
#if defined(_LP64)
        uint_t          p_sharepad;     /* pad for growing p_share */
#endif
        uint_t          p_slckcnt;      /* number of softlocks */
#if defined(__sparc)
        uint_t          p_kpmref;       /* number of kpm mapping sharers */
        struct kpme     *p_kpmelist;    /* kpm specific mapping info */
#else
        /* index of entry in p_map when p_embed is set */
        uint_t          p_mlentry;
#endif
#if defined(_LP64)
        kmutex_t        p_ilock;        /* protects p_vpmref */
#else
        uint64_t        p_msresv_2;     /* page allocation debugging */
#endif
} page_t;
```

______***unchanged_portion_omitted_***