

Addendum to Swapping and Page Fault Handling

Teming Kuo, 6 May 2002.
NCTU CIS Operating System lab.
2002 Linux kernel trace seminar

Outline

- | Page cache and swap cache
- | Shrink_caches in kswapd
- | Try_to_unuse in Sys_swapoff
- | Do_swap_page in handle_pte_fault

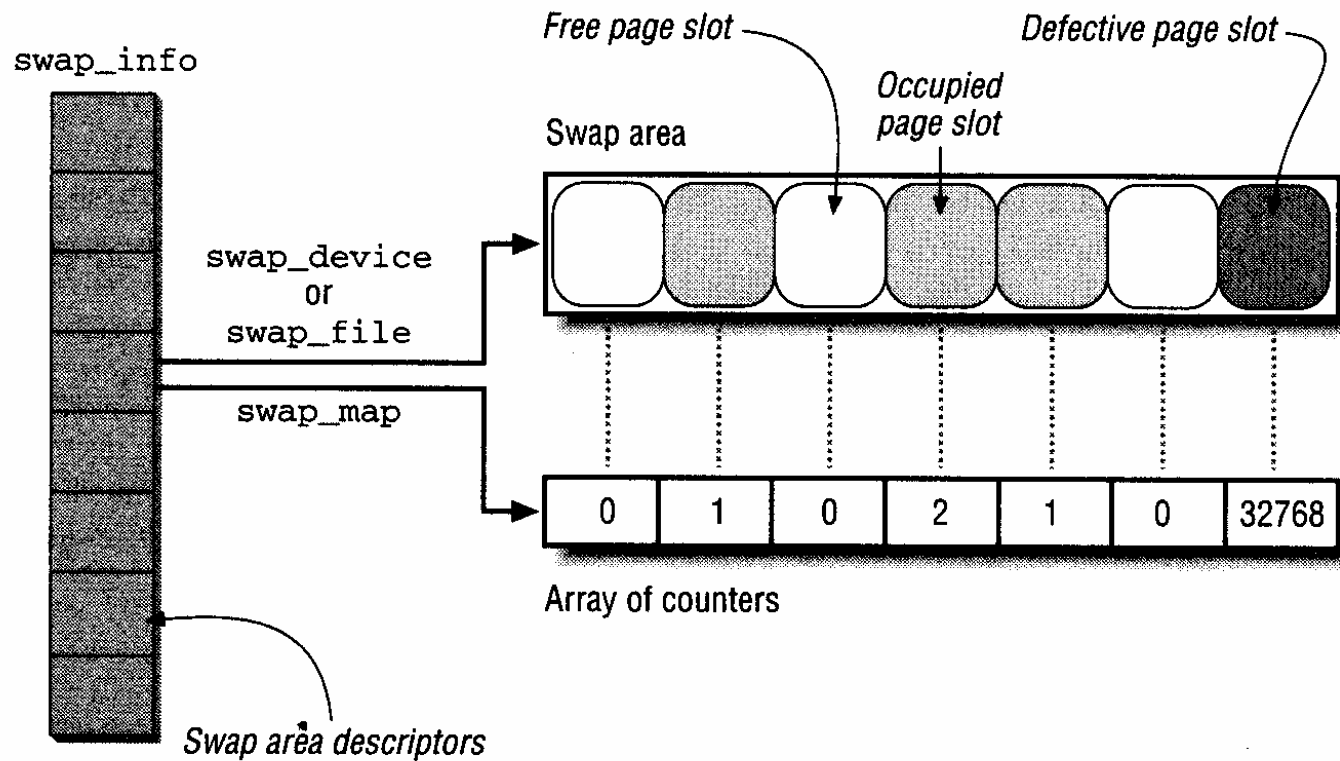
Page Cache and swap cache

- | Disk cache for the accessed by page I/O operations
- | A page in the page cache is identified by a file's inode and by the offset within the file
- | It makes use of 2 main data structures
 - A page hash table
 - An inode queue
- | Page lists which collectively comprise the page cache
 - Active_list
 - Inactive_list, pages not referenced since last scan

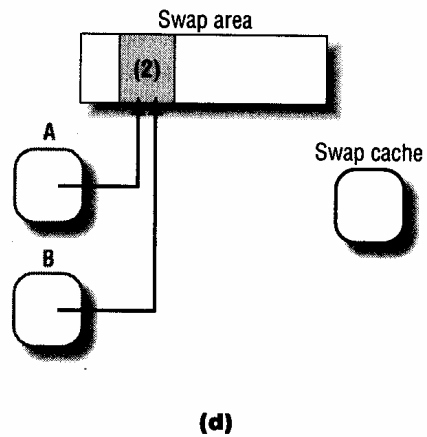
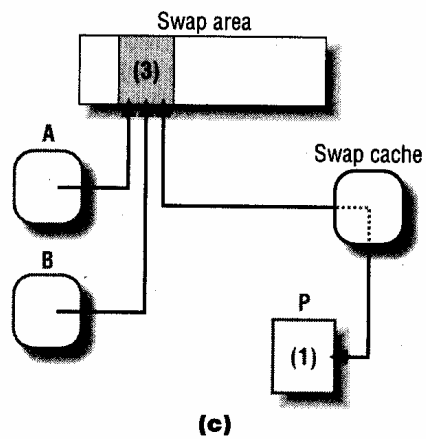
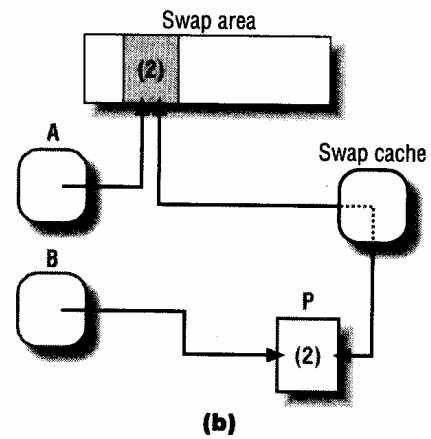
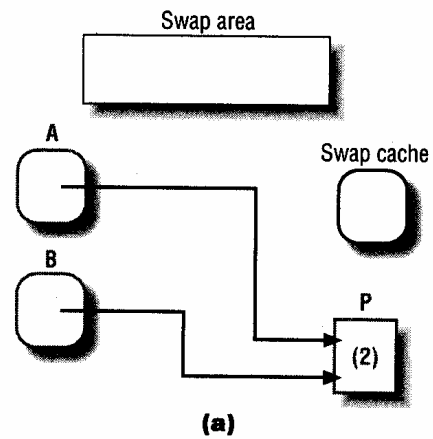
Swap cache

- | The swap cache is just the part of the page cache associated with swap devices.
- | Anonymous pages don't get added to the swap cache until the first time they are evicted from a process's memory map.
- | It's used to collect all shared page frames that have been copied to swap areas.
- | Special treatment compared with page cache
 - *mapping* field of the page descriptor stores the address of a fictitious inode object, `swapper_space`
 - *offset* field stores the swapped-out page identifier associated with the page.

Swap Area Data Structures



The role of the swap cache



kswapd

```
int kswapd(void *unused)
{
    ...
    for (;;) {
        __set_current_state(TASK_INTERRUPTIBLE);
        add_wait_queue(&kswapd_wait_queue, &current);
        mb();
        if (kswapd_can_sleep())
            schedule();

        __set_current_state(TASK_RUNNING);
        remove_wait_queue(&kswapd_wait_queue, &current);

        kswapd_balance();
        run_task_queue(&tq_disk);
    }
}
```

```
static int kswapd_can_sleep(void)
{
    pg_data_t * pgdat;
    pgdat = pgdat_list;
    do {
        if (kswapd_can_sleep_pgdat(pgdat))
            continue;
        return 0;
    } while ((pgdat = pgdat->node_next));
    return 1;
}
```

```
static void kswapd_balance(void)
{
    int need_more_balance;
    pg_data_t * pgdat;

    do {
        need_more_balance = 0;
        pgdat = pgdat_list;
        do
            need_more_balance |= kswapd_balance_pgdat(pgdat);
        while ((pgdat = pgdat->node_next));
    } while (need_more_balance);
}
```

Kswapd_balance_pgdat

```
static int kswapd_balance_pgdat(pg_data_t * pgdat)
{
    int need_more_balance = 0, i;
    zone_t * zone;
    for (i = pgdat->nr_zones-1; i >= 0; i--) {
        zone = pgdat->node_zones + i;
        ...
        if (!zone->need_balance)
            continue;
        if (!try_to_free_pages(zone, GFP_KSWAPD,
            zone->need_balance = 0;
            __set_current_state(TASK_INTERRUPTIBLE);
            schedule_timeout(HZ);
            continue;
        }
        if (check_classzone_need_balance(zone))
            need_more_balance = 1;
        else
            zone->need_balance = 0;
    }
    return need_more_balance;
}

int try_to_free_pages(zone_t
*classzone, unsigned int gfp_mask,
unsigned int order)
{
    int priority = DEF_PRIORITY;
    int nr_pages = SWAP_CLUSTER_MAX;
    gfp_mask =
        pf_gfp_mask(gfp_mask);
    do {
        nr_pages =
shrink_caches(classzone, priority,
            gfp_mask, nr_pages);
        if (nr_pages <= 0)
            return 1;
    } while (--priority);
    out_of_memory();
    return 0;
}
```

shrink_caches

```
static int shrink_caches(zone_t * classzone, int priority,
                        int nr_pages)
{
    int chunk_size = nr_pages;
    unsigned long ratio;
    nr_pages -= kmem_cache_reap(gfp_mask);
    if (nr_pages <= 0)
        return 0;
    nr_pages = chunk_size;
    /* try to keep the active list 2/3 of total */
    ratio = (unsigned long) nr_pages * 2 / 3;
    refill_inactive(ratio);
    nr_pages = shrink_cache(nr_pages, classzone);
    if (nr_pages <= 0)
        return 0;
    shrink_dcache_memory(priority, gfp_mask);
    shrink_icache_memory(priority, gfp_mask);
#ifdef CONFIG_QUOTA
    shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
#endif
    return nr_pages;
}
```

```
static void refill_inactive(int nr_pages)
{
    ...
    entry = active_list.prev;
    while (nr_pages-- && entry != &active_list) {
        struct page * page;
        page = list_entry(entry, struct page, lru);
        entry = entry->prev;
        if (PageTestandClearReferenced(page)) {
            list_del(&page->lru);
            list_add(&page->lru, &active_list);
            continue;
        }
        del_page_from_active_list(page);
        add_page_to_inactive_list(page);
        SetPageReferenced(page);
    }
    spin_unlock(&pagemap_lru_lock);
}
```

```
int shrink_dcache_memory(int priority, unsigned int gfp_mask)
{
    ...
    count = dentry_stat.nr_unused / priority;
    prune_dcache(count);
    kmem_cache_shrink(dentry_cache);
    return 0;
}
```

```
int shrink_icache_memory(int priority, int gfp_mask)
{
    ...
    count = inodes_stat.nr_unused / priority;
    prune_icache(count);
    kmem_cache_shrink(inode_cachep);
    return 0;
}
```

Swap_out

```
static int swap_out(unsigned int priority,
{
    int counter, nr_pages = SWAP_CLUSTER_M
    struct mm_struct *mm;
    counter = mmlist_nr;
    do {
        ... mm = swap_mm;
        while (mm->swap_address == TASK
            mm->swap_address = 0;
            mm = list_entry(mm->mm
                if (mm == swap_mm)
                    swap_mm = mm;
        }
        ...
        nr_pages = swap_out_mm(mm, nr_pages, &counter, classzone);
        mmput(mm);
        if (!nr_pages)
            return 1;
    } while (--counter >= 0);
    return 0;
    ...
}
```

```
static inline int swap_out_mm(struct mm_struct * mm, int count, int *
mmcounter, zone_t * classzone)
{
    ...
    vma = find_vma(mm, address);
    if (vma) {
        if (address < vma->vm_start)
            address = vma->vm_start;
        for (;;) {
            count = swap_out_vma(mm, vma, address, count, classzone);
            vma = vma->vm_next;
            if (!vma)
                break;
            if (!count)
                goto out_unlock;
            address = vma->vm_start;
        }
    }
    mm->swap_address = TASK_SIZE;
    ...
    return count;
}
```

Swap_out_pmd

```
static inline int swap_out_pmd(struct mm_struct * mm, struct vm_area_struct * vma,
    pmd_t *dir, unsigned long address, unsigned long end, int count, zone_t * classzone)
{
    ...
    pte = pte_offset(dir, address);
    pmd_end = (address + PMD_SIZE) & PMD_MASK;
    if (end > pmd_end)        end = pmd_end;
    do {
        if (pte_present(*pte)) {
            struct page *page = pte_page(*pte);
            if (VALID_PAGE(page) && !PageReserved(page)) {
                count -= try_to_swap_out(mm, vma, address, pte, page, classzone);
                if (!count) {
                    address += PAGE_SIZE;
                    break;
                }
            }
        }
        address += PAGE_SIZE; pte++;
    } while (address && (address < end));
    mm->swap_address = address;
    return count;
}
```

try_to_swap_out—(1)

```
static inline int try_to_swap_out(struct mm_struct * mm, struct vm_area_struct* vma,
    unsigned long address, pte_t * page_table, struct page *page, zone_t * classzone)
{
    ...
    if ((vma->vm_flags & VM_LOCKED) || ptep_test_and_clear_young(page_table)) {
        mark_page_accessed(page);
        return 0;
    }
    /* Don't bother unmapping pages that are active */
    if (PageActive(page))
        return 0;
    /* Don't bother replenishing zones not under pressure.. */
    if (!memclass(page->zone, classzone))
        return 0;
    if (TryLockPage(page))
        return 0;
    flush_cache_page(vma, address);
    pte = ptep_get_and_clear(page_table);
    flush_tlb_page(vma, address);
    if (pte_dirty(pte))
        set_page_dirty(page);
}
```

```
static inline pte_t ptep_get_and_clear(pte_t *ptep)
{
    pte_t res;
    /* xchg acts as a barrier before the setting of the high bits */
    res.pte_low = xchg(&ptep->pte_low, 0);
    res.pte_high = ptep->pte_high;
    ptep->pte_high = 0;
    return res;
}
```

try_to_swap_out—(2)

```
    if (PageSwapCache(page)) {
        entry.val = page->index;
        swap_duplicate(entry);
set_swap_pte:
        set_pte(page_table, swp_entry_to_pte(entry));
drop_pte:
        mm->rss--;
        UnlockPage(page);
        {
            int freeable = page_count(page) - !!page->buffers <= 2;
            page_cache_release(page);
            return freeable;
        }
    }
    if (page->mapping)
        goto drop_pte;
    if (!PageDirty(page))
        goto drop_pte;
    ...
}

    for (;;) {
        entry = get_swap_page();
        if (!entry.val)
            break;
        if (add_to_swap_cache(page, entry) == 0) {
            SetPageUptodate(page);
            set_page_dirty(page);
            goto set_swap_pte;
        }
        swap_free(entry);
    }
}
```

Sys_swapoff—(1)

```
asmlinkage long sys_swapoff(const char * specialfile)
{
    ...
    prev = -1;
    swap_list_lock();
    for (type = swap_list.head; type >= 0; type = swap_info[type].next) {
        p = swap_info + type;
        if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
            if (p->swap_file == nd.dentry)
                break;
        }
        prev = type;
    }
    if (type < 0) {
        swap_list_unlock();
        goto out_dput;
    }
}
```

Sys_swapoff—(2)

```
if (prev < 0) {
    swap_list.head = p->next;
} else {
    swap_info[prev].next = p->next;
}
if (type == swap_list.next) {
    /* just pick something that's safe... */
    swap_list.next = swap_list.head;
}
nr_swap_pages -= p->pages;
total_swap_pages -= p->pages;
p->flags = SWP_USED;
swap_list_unlock();
unlock_kernel();
err = try_to_unuse(type);
```

Try_to_unuse—(1

```
static int try_to_unuse(unsigned int type)
{
    struct swap_info_struct * si = &swap_i
    ...
    while ((i = find_next_to_unuse(si, i)),
           swap_map = &si->swap_map[i];
           entry = SWP_ENTRY(type, i);
           page = read_swap_cache_async(ent
```

```
#define SWP_ENTRY(type, offset)
((swp_entry_t) { ((type) << 1) | ((offset) << 8) })
```

```
    lock_page(page);
    swcount = *swap_map;
    if (swcount > 1) {
        flush_page_to_ram(page);
        if (start_mm == &init_mm)
            shmem_unuse(entry, page);
        else
            unuse_process(start_mm, entry);
    }
}
```

```
static int find_next_to_unuse(struct swap_info_struct *si, int prev)
{
    int max = si->max;
    int i = prev;
    int count;

    for (;;) {
        if (++i >= max) {
            if (!prev) {
                i = 0;
                break;
            }
            max = prev + 1;
            prev = 0;
            i = 1;
        }
        count = si->swap_map[i];
        if (count && count != SWAP_MAP_BAD)
```

```
struct page * read_swap_cache_async(swp_entry_t entry)
{
    ....
    do {
        found_page = find_get_page(&swapper_space, entry.val);
        if (found_page)
            break;
        if (!new_page) {
            new_page = alloc_page(GFP_HIGHUSER);
            if (!new_page)
                break; /* Out of memory */
        }
        err = add_to_swap_cache(new_page, entry);
        if (!err) {
            rw_swap_page(READ, new_page);
            return new_page;
        }
    } while (err != -ENOENT);
    if (new_page)
        page_cache_release(new_page);
    return found_page;
}
```

unuse_process

```
static void unuse_process(struct mm_struct * mm,  
                          swp_entry_t entry, struct page* page)
```

```
static void unuse_vma(struct vm_area_struct * vma, pgd_t *pgdir,  
                      swp_entry_t entry, struct page* page)  
{  
    unsigned long start = vma->vm_start, end = vma->vm_end;  
    if (start >= end)  
        BUG();  
    do {  
        unuse_pgd(vma, pgdir, start, end - start, entry, page);  
        start = (start + PGDIR_SIZE) & PGDIR_MASK;  
        pgdir++;  
    } while (start && (start < end));  
}
```

```
static inline void unuse_pte(struct vm_area_struct * vma, unsigned long  
address, pte_t *dir, swp_entry_t entry, struct page* page)  
{  
    ...  
    get_page(page);  
    set_pte(dir, pte_mkold(mk_pte(page, vma->vm_page_prot)));  
    swap_free(entry);  
    ++vma->vm_mm->nr_swap_pages;  
}
```

```
static int swap_entry_free(struct swap_info_struct *p, unsigned long offset)  
{  
    int count = p->swap_map[offset];  
    if (count < SWAP_MAP_MAX) {  
        count--;  
        p->swap_map[offset] = count;  
        if (!count) {  
            if (offset < p->lowest_bit)  
                p->lowest_bit = offset;  
            if (offset > p->highest_bit)  
                p->highest_bit = offset;  
            nr_swap_pages++;  
        }  
    }  
    return count;  
}
```

```
    lock);  
    for (vma = mm->mmap; vma; vma = vma->vm_next)  
        unuse_vma(vma, pgd, entry, page);  
    spin_unlock(&mm->page_table_lock);  
    return;  
}
```

Try_to_unuse—(2)

```
if (*swap_map > 1) {
    int set_start_mm = (*swap_map >= swcount);
    struct list_head *p = &start_mm->mmlist;
    struct mm_struct *new_start_mm = start_mm;
    ...
    while (*swap_map > 1 && (p = p->next) != &start_mm->mmlist) {
        mm = list_entry(p, struct mm_struct, mmlist);
        swcount = *swap_map;
        if (mm == &init_mm) {
            set_start_mm = 1;
            shmem_unuse(entry, page);
        } else
            unuse_process(mm, entry, page);
        if (set_start_mm && *swap_map < swcount) {
            new_start_mm = mm;
            set_start_mm = 0;
        }
    }
    ...
    mmput(start_mm);
    start_mm = new_start_mm;
}
```

do_swap_page

```
static int do_swap_page(struct mm_struct * mm,
                        struct vm_area_struct * vma,
                        unsigned long address,
                        unsigned long * pte_address,
                        int * ret)
{
    struct page * lookup_swap_cache(swp_entry_t entry) {
        found = find_get_page(&swapper_space, entry.val);
        return found;
    }

    page = lookup_swap_cache(entry);
    if (!page) {
        swpin_readahead(entry);
        page = read_swap_cache_async(entry);
        ...
    }
    mark_page_accessed(page);
    #define vm_swap_full() (nr_swap_pages*2 < total_swap_pages)
    swap_free(entry);
    if (vm_swap_full())
        remove_exclusive_swap_page(entry);
    mm->rss++;
    pte = mk_pte(page, vma->vm_page_prot);
    if (write_access && can_share_swap_page(vma, pte))
        pte = pte_mkdirty(pte_mkwrite(pte));
    ...
    set_pte(page_table, pte);
}

int remove_exclusive_swap_page(struct page *page)
{
    ...
    entry.val = page->index;
    p = swap_info_get(entry);
    if (!p)
        return 0;
    retval = 0;
    if (p->swap_map[SWP_OFFSET(entry)] == 1) {
        spin_lock(&pagecache_lock);
        if (page_count(page) - !!page->buffers == 2) {
            __delete_from_swap_cache(page);
            SetPageDirty(page);
            retval = 1;
        }
        spin_unlock(&pagecache_lock);
    }
    swap_info_put(p);
    if (retval) {
        block_flushpage(page, 0);
        swap_free(entry);
        page_cache_release(page);
    }
    return retval;
}
```

Reference

- | **Linux Core Kernel Commentary**
second edition
- | **Understanding the LINUX KERNEL**
O'reilly
- | **Cross-Referencing Linux**
– <http://lxr.linux.no/>