


Swapping and Page Fault Handling

Teming Kuo, 29 Apr 2002.
NCTU CIS Operating System lab.
2002 Linux kernel trace seminar



Outline

- | Swapping
- | Page fault handling

Swapping

Purposes

- I To expand the address space that is effectively usable by a process
- I To expand the amount of dynamic RAM to load processes

Issues to Be Considered

- | Which kind of page to be swapped out
- | How to distribute pages in the swap area
- | How to select the page to be swapped out
- | When to perform page swap-out

Which kind of page to be swapped out

- | Pages belonging to an anonymous memory region of a process
- | Modified page belonging to a private memory mapping of a process
- | Pages belongs to an IPC shared memory region

How to Distribute Pages In the Swap Area

- | Each *slot* contains exactly one page.
- | Faster swap areas get a higher priority.
- | Swap areas of the same priority are cyclically selected to avoid overloading one of them

How to Select the Page to Be Swapped Out

- | The process that performed fewer page faults is selected to reclaim memory.
- | Accessed flag included in each page table entry to simulate LRU algorithm

When to Perform Page Swap-out

- I The kernel thread, *kswapd*, activated once every second whenever the number of free pages frames falls below a predefined threshold
- I When a memory request to the Buddy system cannot be satisfied

Swap Area Descriptor

```
struct swap_info_struct {
    unsigned int flags;
    kdev_t swap_device;
    spinlock_t sdev_lock;
    struct dentry * swap_file;
    struct vfsmount *swap_vfsmnt;
    unsigned short * swap_map;
    unsigned int lowest_bit;
    unsigned int highest_bit;
    unsigned int cluster_next;
    unsigned int cluster_nr;
    int prio;
    int pages;
    unsigned long max;
    int next;
};
```

`#define SWP_USED 1`
`#define SWP_WRITEOK 3`

first page slot to be scanned when searching for a free one

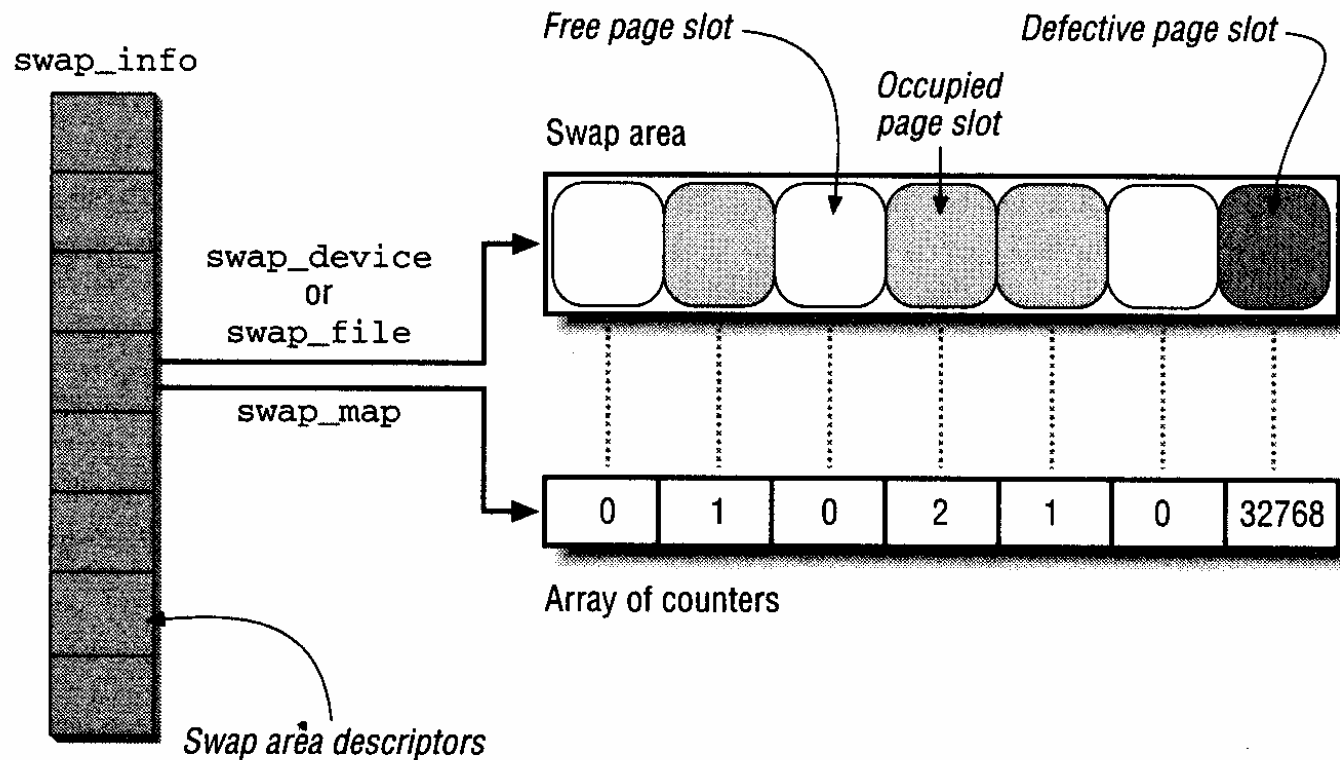
last page slot to be scanned when searching for a free one

next page slot to be scanned when searching for a free one

Number of free page slot allocations before restarting from beginning

`/* next entry on swap list */`

Swap Area Data Structures



Sys_swapon —(1)

```
asmlinkage long sys_swapon(const char * specialfile, int swap_flags)
{
    ...
    swap_list_lock();
    p = swap_info;
    for (type = 0 ; type < nr_swapfiles ; type++,p++)
        if (!(p->flags & SWP_USED))
            break;
    error = -EPERM;
    if (type >= MAX_SWAPFILES) {
        swap_list_unlock();
        goto out;
    }
    if (type >= nr_swapfiles)
        nr_swapfiles = type+1;
    ...
    if (swap_flags & SWAP_FLAG_PREFER)
        p->prio = (swap_flags & SWAP_FLAG_PRIO_MASK)>>SWAP_FLAG_PRIO_SHIFT;
    else
        p->prio = --least_priority;
    swap_list_unlock();
    error = user_path_walk(specialfile, &nd);
}
```

#define MAX_SWAPFILES 32

#define SWAP_FLAG_PREFER 0x8000
#define SWAP_FLAG_PRIO_MASK 0x7fff
#define SWAP_FLAG_PRIO_SHIFT 0

Sys_swapon —(2)

```
swap_header = (void *) __get_free_page(GFP_USER);
if (!swap_header) {
    ...
    goto bad_swap;
}
lock_page(virt_to_page(swap_header));
rw_swap_page_nolock(READ, SWP_ENTRY(type,0), (char *) swap_header);

if (!memcmp("SWAP-SPACE", swap_header->magic.magic, 10))
    swap_header_version = 1;
else if (!memcmp("SWAPSPACE2", swap_header->magic.magic, 10))
    swap_header_version = 2;
else {
    printk("Unable to find swap-space signature\n");
    error = -EINVAL;
    goto bad_swap;
}
```

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

Sys_swapon —(3)

```
switch (swap_header_version) {
case 1:
    memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
    j = 0;p->lowest_bit = 0;p->highest_bit = 0;
    for (i = 1 ; i < 8*PAGE_SIZE ; i++)
        if (test_bit(i,(char *) swap_header)) {
            if (!p->lowest_bit)
                p->lowest_bit = i;
            p->highest_bit = i;
            maxpages = i+1;
            j++;
        }
    nr_good_pages = j;
    p->swap_map = vmalloc(maxpages * sizeof(short));
    ...
    for (i = 1 ; i < maxpages ; i++)
        if (test_bit(i,(char *) swap_header))
            p->swap_map[i] = 0;
        else
            p->swap_map[i] = SWAP_MAP_BAD;
```

```
union swap_header {
    struct
    {
        char reserved[PAGE_SIZE - 10];
        char magic[10];
    } magic;
    struct
    {
        char bootbits[1024];
        unsigned int version;
        unsigned int last_page;
        unsigned int nr_badpages;
        unsigned int padding[125];
        unsigned int badpages[1];
    } info;
};
```

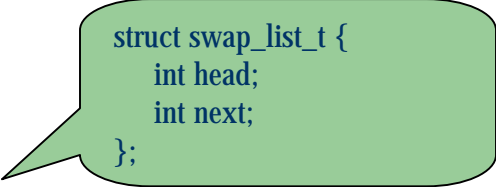
```
#define SWAP_MAP_MAX 0x7fff
#define SWAP_MAP_BAD 0x8000
```

Sys_swapon —(4)

```
switch (swap_header_version) {
case 2:
    ...
    if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
        error = -ENOMEM;
        goto bad_swap;
    }
    memset(p->swap_map, 0, maxpages * sizeof(short));
    for (i=0; i<swap_header->info.nr_badpages; i++) {
        int page = swap_header->info.badpages[i];
        if (page <= 0 || page >= swap_header->info.last_page)
            error = -EINVAL;
        else
            p->swap_map[page] = SWAP_MAP_BAD;
    }
    nr_good_pages = swap_header->info.last_page -
        swap_header->info.nr_badpages -
        1 /* header page */;
}
```

Sys_swapon —(5)

```
/* insert swap space into swap_list: */
prev = -1;
for (i = swap_list.head; i >= 0; i = swap_info[i].next) {
    if (p->prio >= swap_info[i].prio) {
        break;
    }
    prev = i;
}
p->next = i;
if (prev < 0) {
    swap_list.head = swap_list.next = p - swap_info;
} else {
    swap_info[prev].next = p - swap_info;
}
swap_device_unlock(p);
swap_list_unlock();
error = 0;
```



```
struct swap_list_t {
    int head;
    int next;
};
```

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);
```

Sys_swapoff—(3)

```
if (err) {
    /* re-insert swap space back into swap_list */
    swap_list_lock();
    for (prev = -1, i = swap_list.head; i >= 0; prev = i, i =
        swap_info[i].next)
        if (p->prio >= swap_info[i].prio)
            break;

    p->next = i;
    if (prev < 0)
        swap_list.head = swap_list.next = p - swap_info;
    else
        swap_info[prev].next = p - swap_info;
    nr_swap_pages += p->pages;
    total_swap_pages += p->pages;
    p->flags = SWP_WRITEOK;
    swap_list_unlock();
    goto out_dput;
}
...
out_dput:
    ...
}
```

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_KERNEL,
            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();
}
```

```
static int check_classzone_need_balance(zone_t * classzone)
{
    zone_t * first_classzone;

    first_classzone = classzone->zone_pgdat->node_zones;
    while (classzone >= first_classzone) {
        if (classzone->free_pages > classzone->pages_high)
            return 0;
        classzone--;
    }
    return 1;
}
```

Page fault handling

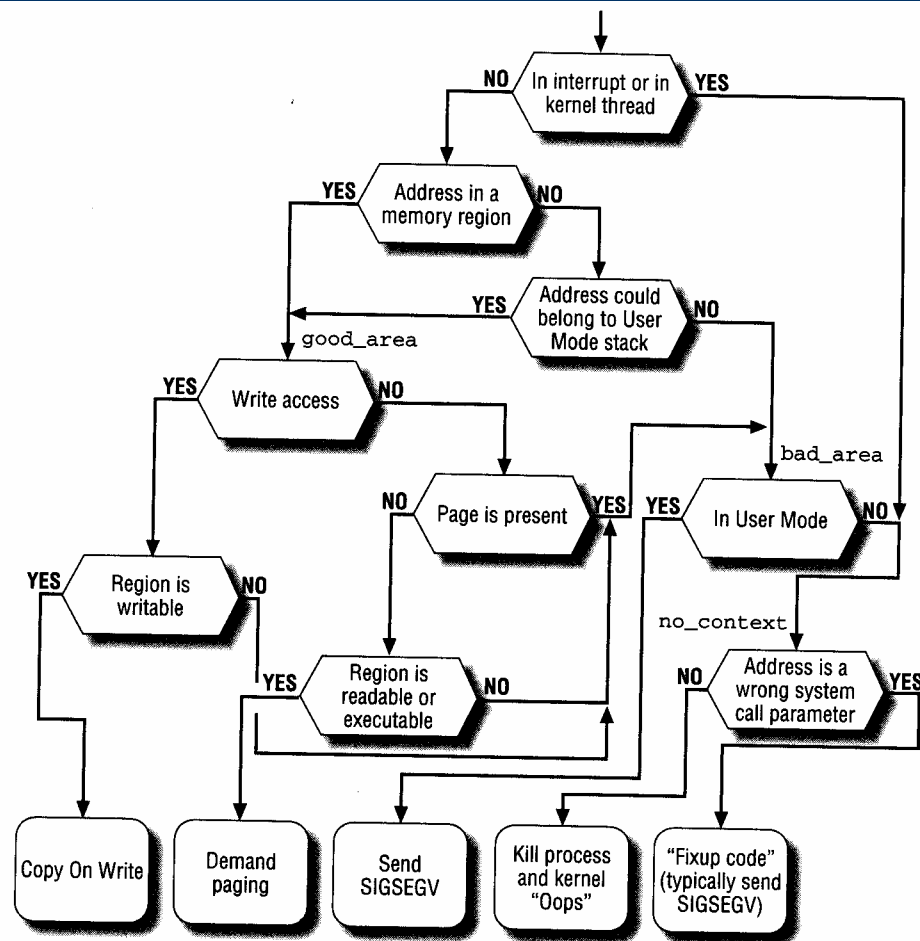
- I Demand paging

Technique that consists of deferring page frame allocation until it is accessed, which is not in RAM, thus causing a “Page fault” exception

- I Copy on Write

When a `fork()` system call is issued, pages are shared between the parent and the child process. Whenever the parent or the child attempts to write into a shared page frame, an exception occurs.

Flow diagram of Page fault handler



Do_page_fault —(1)

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    ...
    /* get the address */
    __asm__("movl %%cr2,%0":"=r" (address));
    ...
    tsk = current;
    if (address >= TASK_SIZE && !(error_code & 5))
        goto vmalloc_fault;
    mm = tsk->mm;
    info.si_code = SEGV_MAPERR;
    if (in_interrupt() || !mm)
        goto no_context;
    down_read(&mm->mmap_sem);
    vma = find_vma(mm, address);
    if (!vma)
        goto bad_area;
    if (vma->vm_start <= address)
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))
        goto bad_area;
}
```

Bit 0: 1:present
Bit 1: 0: read/execute; 1:write
Bit 2: 0:Kernel, 1:User mode

Do_page_fault —(2)

```
    if (expand_stack(vma, address))
        goto bad_area;
good_area:
    info.si_code = SEGV_ACCERR;
    write = 0;
    switch (error_code & 3) {
        default: /* 3: write, present */
                /* fall through */
        case 2: /* write, not present */
                if (!(vma->vm_flags & VM_WRITE))
                    goto bad_area;
                write++;
                break;
        case 1: /* read, present */
                goto bad_area;
        case 0: /* read, not present */
                if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
                    goto bad_area;
    }
```

Do_page_fault —(3)

```
survive:
    switch (handle_mm_fault(mm, vma, address, write)) {
    case 1:
        tsk->min_flt++;
        break;
    case 2:
        tsk->maj_flt++;
        break;
    case 0:
        goto do_sigbus;
    default:
        goto out_of_memory;
    }
    ...
    up_read(&mm->mmap_sem);
    return;
```

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, int write_access)
{
    pgd_t *pgd;
    pmd_t *pmd;

    current->state = TASK_RUNNING;
    pgd = pgd_offset(mm, address);

    pmd = pmd_alloc(mm, pgd, address);

    if (pmd) {
        pte_t *pte = pte_alloc(mm, pmd, address);
        if (pte)
            return handle_pte_fault(mm, vma, address, write_access, pte);
    }
    spin_unlock(&mm->page_table_lock);
    return -1;
}
```

Do_page_fault —(4)

```
bad_area:
    up_read(&mm->mmap_sem);
    /* User mode accesses just cause a SIGSEGV */
    if (error_code & 4) {
        tsk->thread.cr2 = address;
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = 14;
        info.si_signo = SIGSEGV;
        info.si_errno = 0;
        /* info.si_code has been set above */
        info.si_addr = (void *)address;
        force_sig_info(SIGSEGV, &info, tsk);
        return;
    }
    ...
no_context:
    ...
    die("Oops", regs, error_code);
    bust_spinlocks(0);
    do_exit(SIGKILL);
```

handle_pte_fault

```
static inline int handle_pte_fault(struct mm_struct *mm,
    struct vm_area_struct * vma, unsigned long address,
    int write_access, pte_t * pte)
{
    pte_t entry;
    entry = *pte; #define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
    if (!pte_present(entry)) { #define pte_none(x)      (!(x).pte_low)
        if (pte_none(entry))
            return do_no_page(mm, vma, address, write_access, pte);
        return do_swap_page(mm, vma, address, pte, entry, write_access);
    }
    if (write_access) { static inline int pte_write(pte_t pte)
        if (!pte_write(entry)) { return (pte).pte_low & _PAGE_RW; }
            return do_wp_page(mm, vma, address, pte, entry);
        entry = pte_mkdirty(entry); static inline pte_t pte_mkdirty(pte_t pte)
        { (pte).pte_low |= _PAGE_DIRTY; return pte; }
    }
    entry = pte_mkyoung(entry);
    establish_pte(vma, address, pte, entry);
    spin_unlock(&mm->page_table_lock);
    return 1;
}
```

static inline void establish_pte(struct vm_area_struct *vma, unsigned long address, pte_t * page_table, pte_t entry) {
set_pte(page_table, entry);
flush_tlb_page(vma, address);
update_mmu_cache(vma, address, entry);
}

do_no_page

```
static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma,
    unsigned long address, int write_access, pte_t *page_table)
{
    ...
    if (!vma->vm_ops || !vma->vm_ops->nopage)
        return do_anonymous_page(mm, vma, page_table, write_access, address);
    spin_unlock(&mm->page_table_lock);
    new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
    ...
    if (write_access && !(vma->vm_flags & VM_SHARED)) {
        struct page * page = alloc_page(GFP_HIGHUSER);
        if (!page) {
            page_cache_release(new_page);
            return -1;
        }
        copy_user_highpage(page, new_page, address);
        page_cache_release(new_page);
        lru_cache_add(page);
        new_page = page;
    }
    spin_lock(&mm->page_table_lock);
    ...
}
```

do_anonymous_page

```
static int do_anonymous_page(struct mm_struct * mm, struct vm_area_struct * vma,
    pte_t *page_table, int write_access, unsigned long addr)
{
    pte_t entry;
    entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
    if (write_access) {
        ...
        page = alloc_page(GFP_HIGHUSER);
        if (!page) goto no_mem;
        clear_user_highpage(page, addr);
        spin_lock(&mm->page_table_lock);
        if (!pte_none(*page_table)) {
            ...
            return 1;
        }
        ...
        entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
        lru_cache_add(page);
        mark_page_accessed(page);
    }
    set_pte(page_table, entry);
    ...
}
```

do_swap_page

```
static int do_swap_page(struct mm_struct * mm,
    struct vm_area_struct * vma, unsigned long address,
    pte_t * page_table, pte_t orig_pte, int write_access)
{
    ...
    page = lookup_swap_cache(entry);
    if (!page) {
        swapin_readahead(entry);
        page = read_swap_cache_async(entry);
        ...
    }
    mark_page_accessed(page);
    ...
    swap_free(entry);
    if (vm_swap_full())
        remove_exclusive_swap_page(page);
    mm->rss++;
    pte = mk_pte(page, vma->vm_page_prot);
    if (write_access && can_share_swap_page(page))
        pte = pte_mkdirty(pte_mkwrite(pte));
    ...
    set_pte(page_table, pte);
}
```

do_wp_page —(1)

```
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct * vma,
    unsigned long address, pte_t *page_table, pte_t pte)
{
    ...
    #define TryLockPage(page) test_and_set_bit(PG_locked, &(page)->flags)
    if (!TryLockPage(old_page)) {
        int reuse = can_share_swap_page(old_page);
        unlock_page(old_page);
        if (reuse) {
            flush_cache_page(vma, address);
            establish_pte(vma, address, page_table,
                pte_mkyoung(pte_val(pte), old_page));
            spin_unlock(&mm->page_lock);
            return 1; /* Minor fault */
        }
    }
    ...
    new_page = alloc_page(GFP_HIGHUSER);
    if (!new_page)
        goto no_mem;
    copy_cow_page(old_page, new_page, address);
}
```

```
static inline void copy_cow_page(struct page * from, struct page *
    to, unsigned long address)
{
    if (from == ZERO_PAGE(address)) {
        clear_user_highpage(to, address);
        return;
    }
    copy_user_highpage(to, from, address);
}
```

do_wp_page —(2)

```
spin_lock(&mm->page_table_lock);
if (pte_same(*page_table, pte)) {
    if (PageReserved(old_page))
        ++mm->rss;
    break_cow(vma, new_page, address);
    lru_cache_add(new_page);

    /* Free the old page.. */
    new_page = old_page;
}
...
}
```

`#define pte_same(a, b) ((a).pte_low == (b).pte_low)`

`#define PageReserved(page) test_bit(PG_reserved, &(page)->flags)`

```
static inline void break_cow(struct vm_area_struct * vma, struct
page * new_page, unsigned long address,
pte_t *page_table)
{
    flush_page_to_ram(new_page);
    flush_cache_page(vma, address);
    establish_pte(vma, address, page_table,
pte_mkwrite(pte_mkdirty(mk_pte(new_page, vma-
>vm_page_prot))));
}
```

Do_page_fault —(5)

```
out_of_memory:
    up_read(&mm->mmap_sem);
    if (tsk->pid == 1) {
        tsk->policy |= SCHED_YIELD;
        schedule();
        down_read(&mm->mmap_sem);
        goto survive;
    }
    printk("VM: killing process %s\n", tsk->comm);
    if (error_code & 4)
        do_exit(SIGKILL);
    goto no_context;

do_sigbus:
    up_read(&mm->mmap_sem);
    ...
    force_sig_info(SIGBUS, &info, tsk);
    /* Kernel mode? Handle exceptions or die */
    if (!(error_code & 4))
        goto no_context;
    return;
```

Reference

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