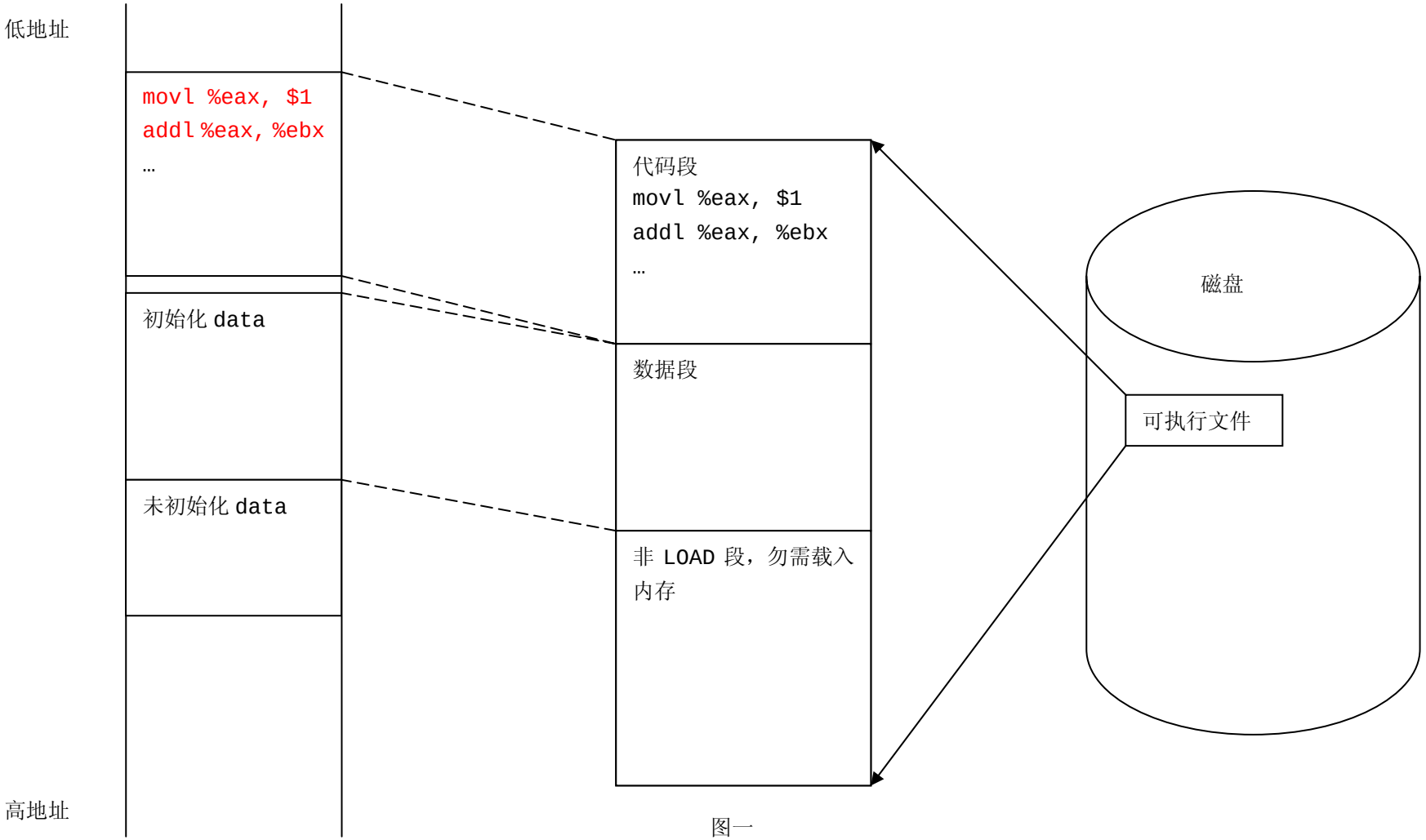
Linux-2.4 内核文件映射的实现解析

前言 2
文件映射介绍 4
源码解析 9
do_mmap_pgoff 源码解析 9
do_page_fault 源码解析 21
handle_mm_fault 源码解析 40
do_no_page()源码解析 46
filemap_nopage 源码解析 51
do_swap_page 源码解析 60
do_anonymous_page 源码解析 66
do_wp_page 源码解析 71
结论 77
联系 78

前言

在《Linux 下 ELF 可执行文件载入（内核部分）源码分析》一文中，我提到在 exec 系统调用结束后，该进程还未返回到用户态以前，内存中并没有任何该可执行文件的代码与数据，有的只是代表该进程与可执行文件之间的映射关系。像下图所示。在图中左边的地址空间中是名副其实是“虚”的，只是一个个空洞而已，物理内存还没有被填入。只有当该进程运行起来时，这些空洞才会被一页页的物理内存填充，有的空洞甚至永远不会被填充。而填充的内容就是对应的磁盘上的可执行文件中的被映射的段。本文分析这个映射过程是怎么建立的。


低地址
movl %eax, $1
addl %eax, %ebx
…
初始化 data
未初始化 data
高地址
代码段
movl %eax, $1
addl %eax, %ebx
…
数据段
非 LOAD 段，勿需载入内存
磁盘
可执行文件

图一

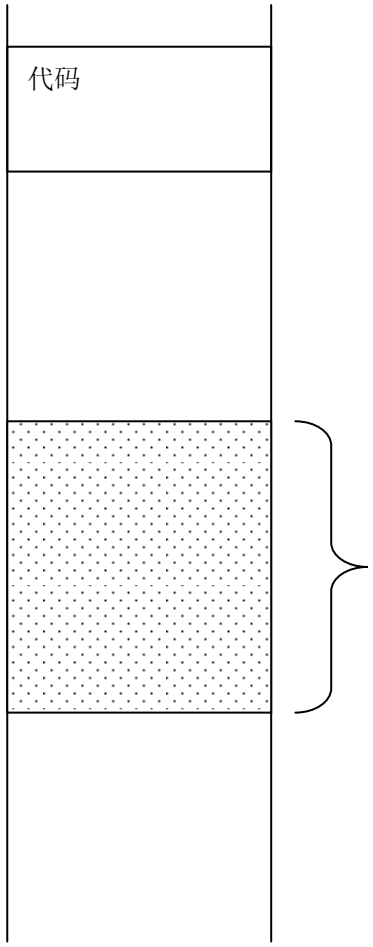
文件映射介绍

文件映射有好多种，上面可执行文件的载入只是其中之一。本文将分析所有的 Linux 中的文件映射类型。

1. 虚拟内存中的某块区域与磁盘上某个特定文件向关联，访问该区域即访问该文件。上面可执行文件的载入就属于此种情况。
2. 虚拟内存中的某块区域没有与任何文件建立关联，只是为了该地址空间而已。这被称为匿名映射。现代 Linux C 库中内存分配函数 malloc 的实现完全依赖于匿名映射。
3. 写时复制（Copy-On-Write, COW）技术的实现完全依赖于文件映射。

上面的图一反映的是上面的第①种情况。而下面的图二与图三分别反映了上面的第②③种情况。

文件映射部分是现代操作系统虚拟地址空间实现的关键，与内核很多模块都有交互，比如 Linux 内核中的 cache 之母 --- Page Cache，比如 swap 空间的管理，我这里只专注于“File Mapping”本身的实现，没有过多牵涉出去。

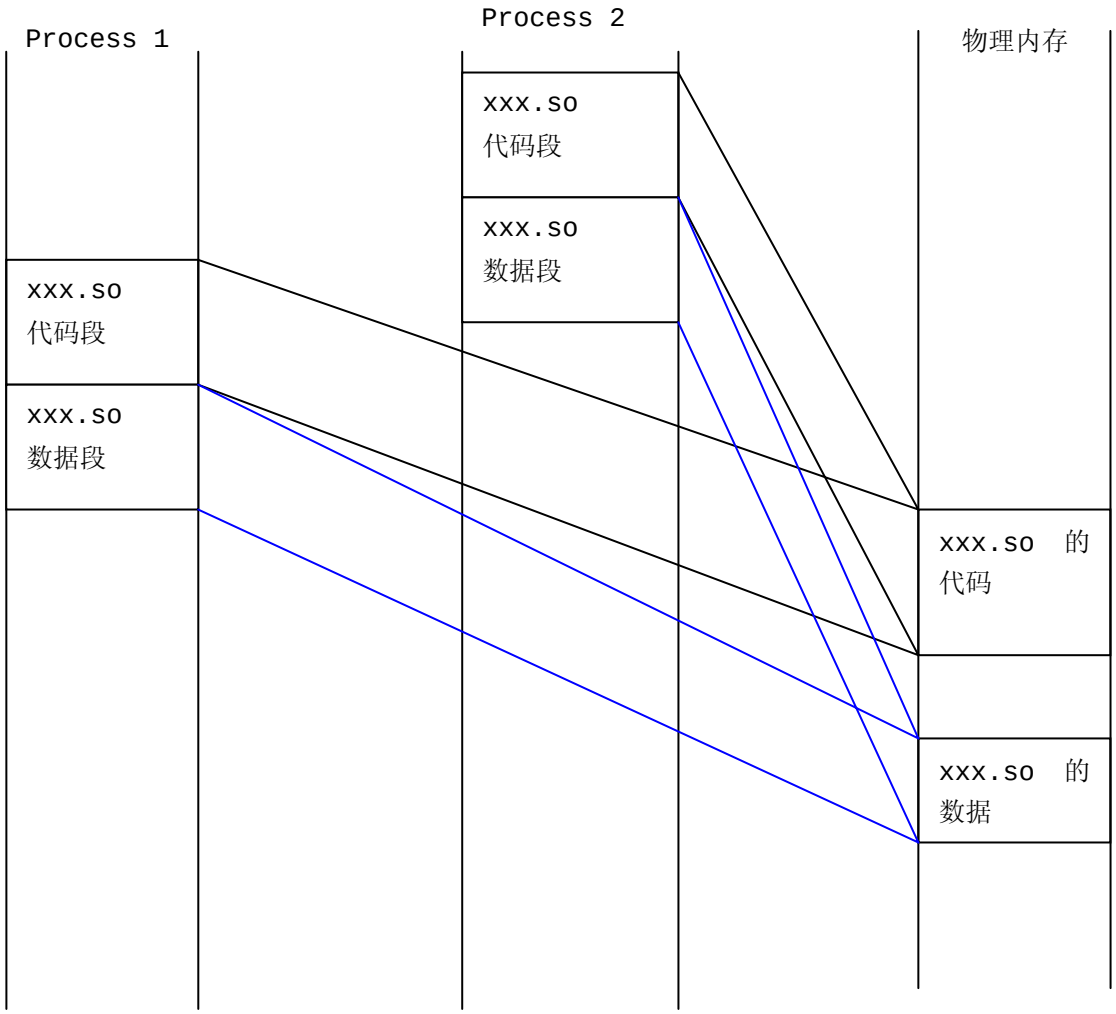


匿名映射

Heap

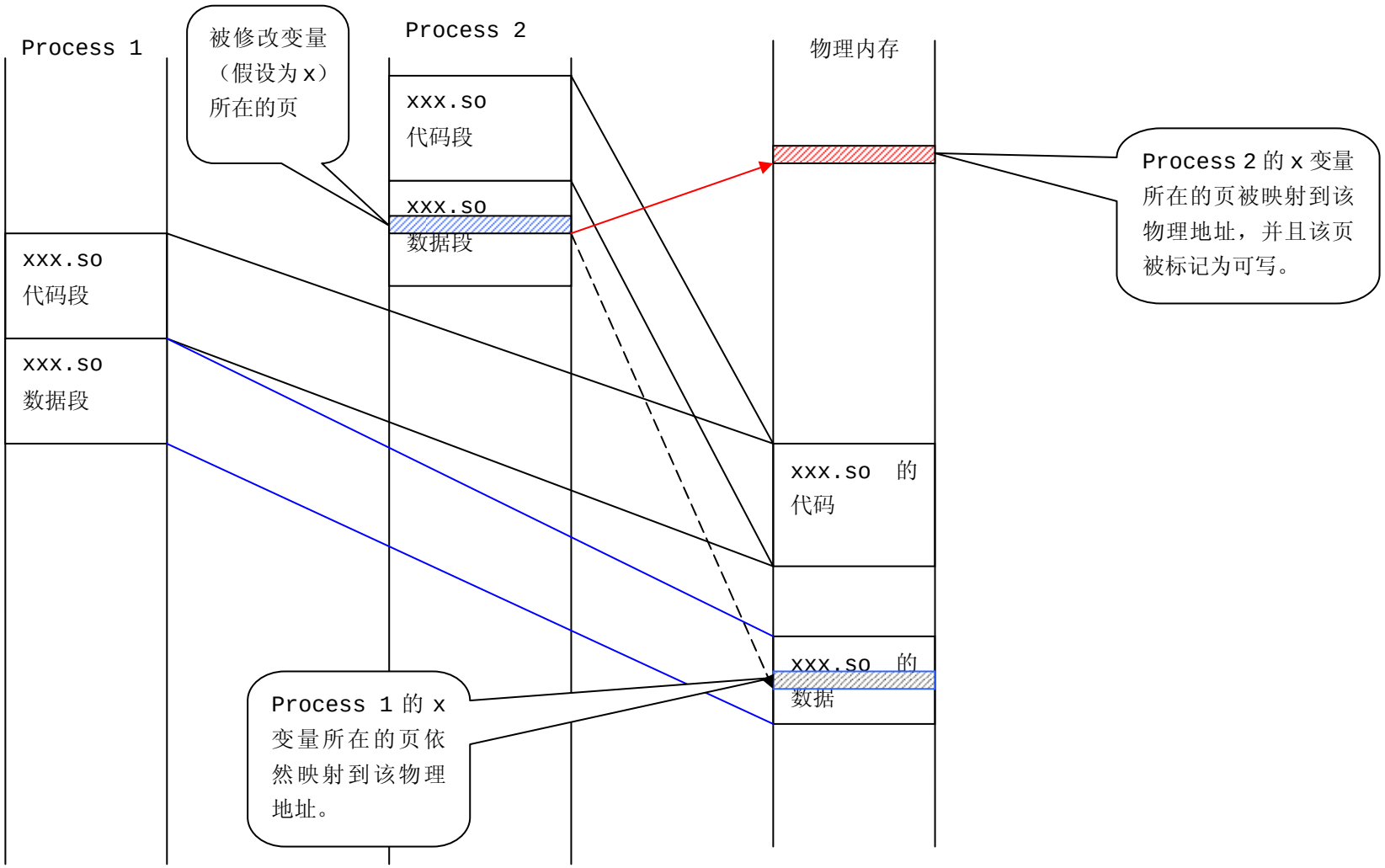
C 库中的内存分配器 malloc 通过 mmap()系统调用来向内核申请大块虚拟空间，然后在此虚拟空间基础上建立自己的分配逻辑与数据结构。这块空间当然不会对应于任何文件。

图二



图三

在初始时，xxx.so 的动态共享库分别被载入 process 1 和 process 2。两个虚拟地址空间的 xxx.so 无论是代码段还是数据段都共享物理内存中的一份拷贝。其中存放代码段与数据段的虚拟空间的页属性都是只读，不可写的。对，即使是数据段也是只读的。但当有某个 process 修改了自己虚拟地址空间中的某个变量后，则该变量所在的页面当然不应该再被共享，否则另一 process 就也看到了该变量的修改。这时就应该用到 Copy-On-Write 技术。


Process 1
被修改变量（假设为x）所在的页
Process 2
物理内存
xxx.so 代码段
xxx.so 数据段
xxx.so 代码段
xxx.so 数据段
xxx.so 的代码
xxx.so 的数据
Process 2 的 x 变量所在的页被映射到该物理地址，并且该页被标记为可写。
Process 1 的 x 变量所在的页依然映射到该物理地址。

图四

无论上面哪一种情况，都会调用文件映射的核心函数 do_mmap_pgoff（），在该函数中为在后面真正建立虚拟页面与物理页面的映射关系而准备。真正建立实打实的映射关系是通过 page fault handler（缺页中断处理器，在 x86 CPU 上为 int 0EH）实现的。即只有在必须要建立映射时才会去花时间去做，也就是所谓的 demanded paging，按需调页。如果不熟悉该功能的理论，请你去找本操作系统的教课书复习一下吧。实际上我觉得，如果你是计算机系科班出身，那下面的代码真应该在操作系统理论课上读。

源码解析

下面注释了与文件映射相关的相关的核心函数。

do_mmap_pgoff 源码解析

```
src/linux-2.4.20/mm/mmap.c

file 为要映射的文件，如果 file = NULL，那么就是匿名映射
addr 为推荐映射的地址
len 为要映射的长度。

unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
    unsigned long prot, unsigned long flags, unsigned long pgoff)
{
    struct mm_struct * mm = current->mm;
    struct vm_area_struct * vma, * prev;
    unsigned int vm_flags;
    int correct_wcount = 0;
    int error;
    rb_node_t ** rb_link, * rb_parent;
```

```
    if (file && (!file->f_op || !file->f_op->mmap))
        return -ENODEV;
```
文件所在文件系统必须支持“mmap”操作。

```
    if ((len = PAGE_ALIGN(len)) == 0)
        return addr;
```
对齐在页边界。

```
    if (len > TASK_SIZE)
        return -EINVAL;
```
要映射的长度竟然超过 3G，当然错。

```
    /* offset overflow? */
    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
        return -EINVAL;
```
pgoff 为文件内页偏移，溢出。

```
    /* Too many mappings? */
    if (mm->map_count > max_map_count)
        return -ENOMEM;
```
超过进程容许的最大映射块数。

```
    /* Obtain the address to map to. we verify (or select) it and ensure
     * that it represents a valid section of the address space.
     */
    addr = get_unmapped_area(file, addr, len, pgoff, flags);
```

```
    if (addr & ~PAGE_MASK)
        return addr;
```
在虚拟地址空间中找一块放得下要映射空间大小的空闲区域。

```
    /* Do simple checking here so the lower-level routines won't have
     * to. we assume access permissions have been handled by the open
     * of the memory object, so we don't do any here.
     */
    vm_flags = calc_vm_flags(prot,flags) | mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

    /* mlock MCL_FUTURE? */
    if (vm_flags & VM_LOCKED) {
```
VM_LOCKED 表示该块区域必须马上对应到物理内存。

```
        unsigned long locked = mm->locked_vm << PAGE_SHIFT;
```
mm->locked_vm 表示当前进程已经占有的物理内存。

```
        locked += len;
        if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
            return -EAGAIN;
```
现有被 lock 住的内存加上本次的，是否会超出限额。每个 process 对各种资源的占有都有一个限额，其被记录在 task_struct 结构的 rlim[]数组内。

```
    }

    if (file) {
```
file 非空，表示 File Mapping。否则是匿名映射。

```
        switch (flags & MAP_TYPE) {
        case MAP_SHARED:
```
表示该文件映射允许共享
```
            if ((prot & PROT_WRITE) && !(file->f_mode & FMODE_WRITE))
```
如果映射该区域是允许“写”的，但被映射的文件却不允许“写”，错
```
                return -EACCES;

            /* Make sure we don't allow writing to an append-only file.. */
            if (IS_APPEND(file->f_dentry->d_inode) && (file->f_mode & FMODE_WRITE))
                return -EACCES;
```
不允许映射 append only 的文件。这可以理解。“append only”表示只能对该文件在尾部追加操作。
```
            /* make sure there are no mandatory locks on the file. */
            if (locks_verify_locked(file->f_dentry->d_inode))
                return -EAGAIN;
```
确认被映射文件没有被 lock，即没有其他执行线程正排他的占有着该文件。
```
            vm_flags |= VM_SHARED | VM_MAYSHARE;
```
由于是“share”，所以这里设置“share”属性。这里的“VM_”打头的都是 vma 的属性值。
```
            if (!(file->f_mode & FMODE_WRITE))
                vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
```
如果文件不可写，则清除相应属性。remove VM_SHARED ?
```
            /* fall through */
        case MAP_PRIVATE:
```
表示不允许共享。

```
            if (!(file->f_mode & FMODE_READ))
                return -EACCES;
```
被映射文件没有“可读”属性，即该要映射的文件不可读，错，报“cann't access”，即没有权限。
```
            break;

        default:
```
该映射区域要么可以共享，要么私有，两者必居其一。
```
            return -EINVAL;
        }
    } else {
```
对匿名区域的映射，也就是该映射只是为了分配一块虚拟地址空间。
```

        vm_flags |= VM_SHARED | VM_MAYSHARE;
        switch (flags & MAP_TYPE) {
        default:
            return -EINVAL;
        case MAP_PRIVATE:
            vm_flags &= ~(VM_SHARED | VM_MAYSHARE);
```
如果该映射是私有的，即不会被多个 process 共享，则清除 share 属性。
```
            /* fall through */
        case MAP_SHARED:
            break;

        }
    }
```

```
    /* Clear old maps */
munmap_back:
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (vma && vma->vm_start < addr + len) {
        if (do_munmap(mm, addr, len))
            return -ENOMEM;
        goto munmap_back;
    }

    /* Check against address space limit. */
    if ((mm->total_vm << PAGE_SHIFT) + len
        > current->rlim[RLIMIT_AS].rlim_cur)
        return -ENOMEM;
```

检查加上本次申请的空间后是否超过额度。

```
    /* Private writable mapping? Check memory availability.. */
    if ((vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
        !(flags & MAP_NORESERVE)            &&
        !vm_enough_memory(len >> PAGE_SHIFT))
        return -ENOMEM;

    /* Can we just expand an old anonymous mapping? */
    if (!file && !(vm_flags & VM_SHARED) && rb_parent)
        if (vma_merge(mm, prev, rb_parent, addr, addr + len, vm_flags))
            goto out;
```

```
    /* Determine the object being mapped and call the appropriate
     * specific mapper. the address has already been validated, but
     * not unmapped, but the maps are removed from the list.
     */
    vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!vma)
        return -ENOMEM;
```

向 slab 分配器申请一个 vma。

```
    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags;
    vma->vm_page_prot = protection_map[vm_flags & 0x0f];
    vma->vm_ops = NULL;
    vma->vm_pgoff = pgoff;
    vma->vm_file = NULL;
    vma->vm_private_data = NULL;
    vma->vm_raend = 0;
```

初始化该 vma。

```
    if (file) {
```

如果为非匿名映射

```
        error = -EINVAL;
        if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
```

```
            goto free_vma;
        if (vm_flags & VM_DENYWRITE) {
            error = deny_write_access(file);
            if (error)
                goto free_vma;
            correct_wcount = 1;
        }
        vma->vm_file = file;
```

把 file 结构纪录在 vma 中。这很重要，在缺页中断中，就是靠它才知道该 page fault 应该对应到哪个文件中。

```
        get_file(file);
```

递增被映射文件的引用计数。

```
        error = file->f_op->mmap(file, vma);
```

该行代码的核心在 ext2 文件系统中只是如下：

```
        vma->vm_ops = &generic_file_vm_ops;
```

对 vma 的 vm_ops 进行赋值。

```
        if (error)
            goto unmap_and_free_vma;
    } else if (flags & MAP_SHARED) {
```

如果是匿名映射，并且又是与其他进程共享则把 vma->vm_file 替换成/dev/zero，并且 vma->vm_ops 设为 shmem_vm_ops。

```
        error = shmem_zero_setup(vma);
        if (error)
            goto free_vma;
```

```
    }

    /* Can addr have changed??
     *
     * Answer: Yes, several device drivers can do it in their
     *         f_op->mmap method. -DaveM
     */
    if (addr != vma->vm_start) {
        /*
         * It is a bit too late to pretend changing the virtual
         * area of the mapping, we just corrupted userspace
         * in the do_munmap, so FIXME (not in 2.4 to avoid breaking
         * the driver API).
         */
        struct vm_area_struct * stale_vma;
        /* Since addr changed, we rely on the mmap op to prevent
         * collisions with existing vmas and just use find_vma_prepare
         * to update the tree pointers.
         */
        addr = vma->vm_start;
        stale_vma = find_vma_prepare(mm, addr, &prev,
                        &rb_link, &rb_parent);
        /*
         * Make sure the lowlevel driver did its job right.
         */
        if (unlikely(stale_vma && stale_vma->vm_start < vma->vm_end)) {
```

```
            printk(KERN_ERR "buggy mmap operation: [<%p>]\n",
                file ? file->f_op->mmap : NULL);
            BUG();
        }
    }

    vma_link(mm, vma, prev, rb_link, rb_parent);
    if (correct_wcount)
        atomic_inc(&file->f_dentry->d_inode->i_writecount);

out:
    mm->total_vm += len >> PAGE_SHIFT;
    if (vm_flags & VM_LOCKED) {
        mm->locked_vm += len >> PAGE_SHIFT;
        make_pages_present(addr, addr + len);
    }
    return addr;

unmap_and_free_vma:
    if (correct_wcount)
        atomic_inc(&file->f_dentry->d_inode->i_writecount);
    vma->vm_file = NULL;
    fput(file);

    /* Undo any partial mapping done by a device driver. */
    zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
```

```
free_vma:
    kmem_cache_free(vm_area_cachep, vma);
    return error;
}
```

```
int generic_file_mmap(struct file * file, struct vm_area_struct * vma)
{
    struct address_space *mapping = file->f_dentry->d_inode->i_mapping;
```

address_space 结构是 page cache 的核心数据结构。这里只简单提一下。文件映射要读写被映射的文件，这“读写”都要通过 page cache，相应的映射文件的处理器不会直接从文件中读出内容或写内容到文件。如果要读取的内容已经在 page cache 中，则直接从 cache 中返回即可；如果 page cache 中没有，则要读取。无论是在 page cache 中寻找有没有需要的文件内容还是读取或写出，都要 address_space 结构来完成。

```
    struct inode *inode = mapping->host;
```

该映射文件的 inode。

```
    if ((vma->vm_flags & VM_SHARED) && (vma->vm_flags & VM_MAYWRITE)) {
        if (!mapping->a_ops->writepage)
```

如果要映射的区域是共享并可写的，但 address_space 结构又没有定义 writepage 操作，就报错。否则就无人负责“写”了。

```
            return -EINVAL;
    }
    if (!mapping->a_ops->readpage)
```

如果连 readpagex 操作都没有，错。

```
        return -ENOEXEC;
    UPDATE_ATIME(inode);     update 该文件的 access time

    vma->vm_ops = &generic_file_vm_ops;
```

设置映射的该区域的操作函数集。当 vma 要完成什么动作时，完全依赖该操作函数集中定义的函数来实现。

```
    return 0;
}
```

do_page_fault 源码解析

```
src/linux-2.4.20/arch/i386/mm/fault.c
```

do_mmap_pgoff()只是为映射的虚拟地址空间与被映射的文件之间建立了关联，主要是对管理映射区域的 vma 设置了对应的值，比如该映射对应到哪个文件；对该区域如何处理等。真正的完成对应是在发生 page fault 之时。即当 CPU 去访问被映射区域，而该区域并没有实际的物理内存相对应，产生 trap 0eh，即缺页中断。内核将在这里完成真正的“映射”--- 为访问的虚拟页面分配物理页面，然后由 vma 记录的信息推断出从某个文件的某处读出内容到刚刚分配的空间中。当然我这里只是举了处理缺页中断的某种场景，非全部。

```
/*
 * This routine handles page faults.  It determines the address,
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 *
 * error_code:
 *  bit 0 == 0 means no page found, 1 means protection fault
 *  bit 1 == 0 means read, 1 means write
 *  bit 2 == 0 means kernel, 1 means user-mode
 */
```

由上面的注释可看到参数 error_code 所带的信息。

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
```

```
    struct vm_area_struct * vma;
    unsigned long address;
    unsigned long page;
    unsigned long fixup;
    int write;
    siginfo_t info;

    /* get the address */
    __asm__("movl %%cr2,%0":"=r" (address));
```

取得产生 page fault 的地址

```
    /* It's safe to allow irq's after cr2 has been saved */
    if (regs->eflags & X86_EFLAGS_IF)
        local_irq_enable();
```

开中断

```
    tsk = current;
```

产生 page fault 时，肯定是在“中断上下文(context)”中。因为在 Linux 内核中的代码都是 lock 在内存里的，它不象 Windows NT，在内核里还分 non-pageable 和 pageable 两种。从好处上来说，当然 Windows NT 内核可以最大限度的节省物理内存，但有两点是值得商榷的：

1. 增加了 driver 开发者的难度，他要么索性把代码全部设成 non-pageable，要么准确地知道哪些代码可以 pageable，而哪些代码可以 non-pageable。
2. 现在 PC 的内存容量非 10 年前可比，少则 512M，多则 1G，而内核加上各种 driver 所占的内存相对于此容量只占很小一部分。在 16/32M 的情况下有必要，在 512/1000M 的情况下是在多次一举。

```
    /*
     * We fault-in kernel-space virtual memory on-demand. The
     * 'reference' page table is init_mm.pgd.
     *
```

```
     * NOTE! We MUST NOT take any locks for this case. We may
     * be in an interrupt or a critical region, and should
     * only copy the information from the master page table,
     * nothing more.
     *
     * This verifies that the fault happens in kernel space
     * (error_code & 4) == 0, and that the fault was not a
     * protection error (error_code & 1) == 0.
     */
    if (address >= TASK_SIZE && !(error_code & 5))
        goto vmalloc_fault;
```

address >= TASK_SIZE 表示该 page fault 发生在内核空间。

error_code & 101

检查 bit 0 和 bit 2，即如果“no page found”且“kernel”

这里的 address >= TASK_SIZE 与“kernel”是同等的条件，要么同时成立，要么同时不成立。

在内核态发生“no page found”，只有是用“vmalloc”分配的内存才可能。

```
    mm = tsk->mm;
    info.si_code = SEGV_MAPERR;

    /*
     * If we're in an interrupt or have no user
     * context, we must not take the fault..
     */
    if (in_interrupt() || !mm)
        goto no_context;
```

如果满足上面的条件满足，表明是在“中断上下文”中。

运行到这里，表示触发 page fault 的地址是用户态的。

```
    down_read(&mm->mmap_sem);
```

要搜索用户态内存映射的红黑树，所以这里为防止竞争，先占有对应的 semaphore。

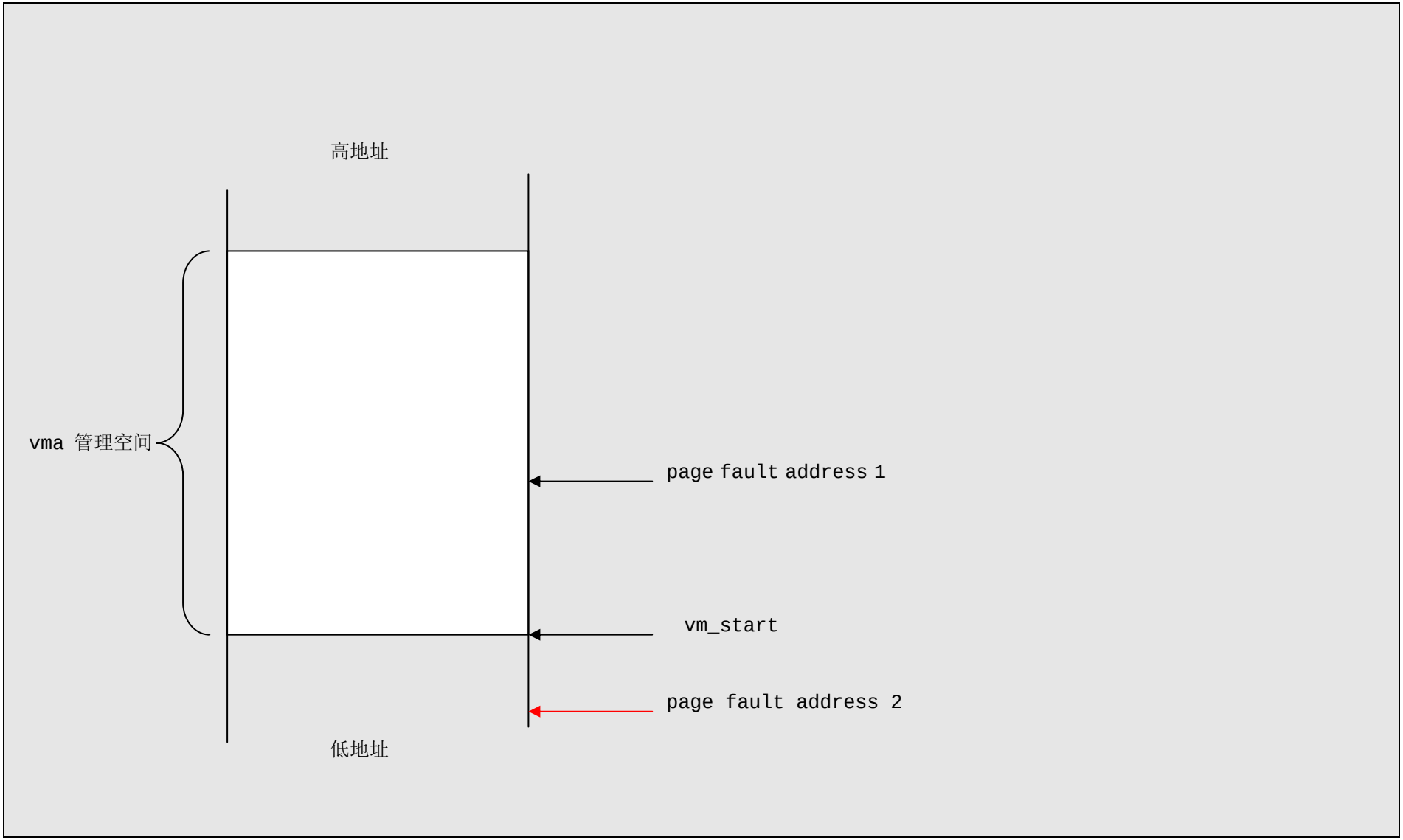
```
    vma = find_vma(mm, address);
    if (!vma)
        goto bad_area;
```

找寻该 page fault address 所在的 vma，即由谁管理该地址。如果每人管该地址，那肯定是非法地址。比如在你的程序里有下面的代码：

```
int *p = 0;
*p = 1;
```

则就会产生 page fault，而零地址又是非法地址。陷入内核后就会运行到这里。

```
    if (vma->vm_start <= address)
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))
        goto bad_area;
```


高地址
vma 管理空间
page fault address 1
vm_start
page fault address 2
低地址

见上图，对于 page fault address 1，自然是合法地址。但对于 page fault address 2，则要看，如果这是常规的 stack，即向下(grow down)增长的，则同样该地址也合法。反之，自然非法。

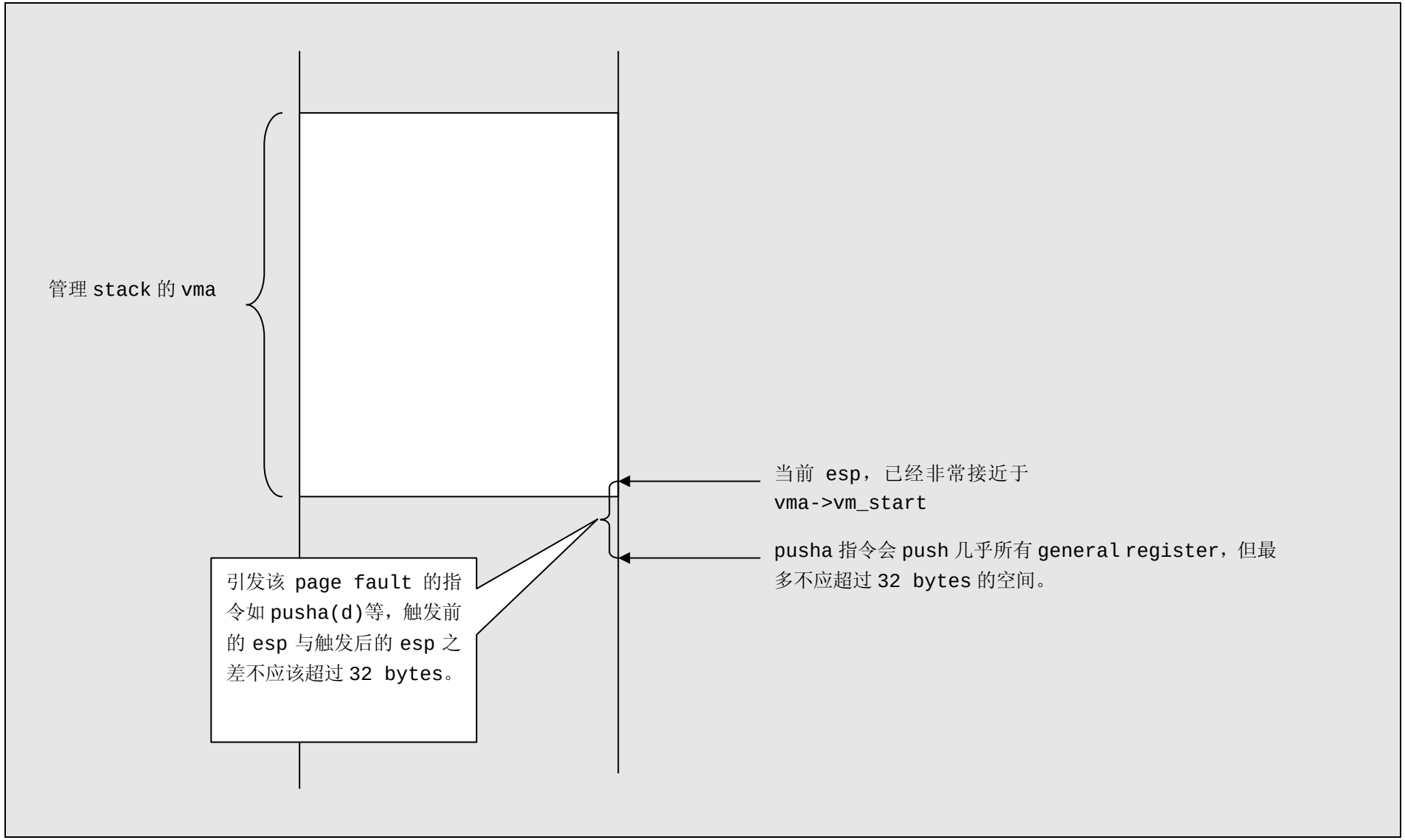
```
   if (error_code & 4) {
```

如果是用户态

```
       /*
        * accessing the stack below %esp is always a bug.
        * The "+ 32" is there due to some instructions (like
        * pusha) doing post-decrement on the stack and that
        * doesn't show up until later..
        */
       if (address + 32 < regs->esp)
          goto bad_area;
```

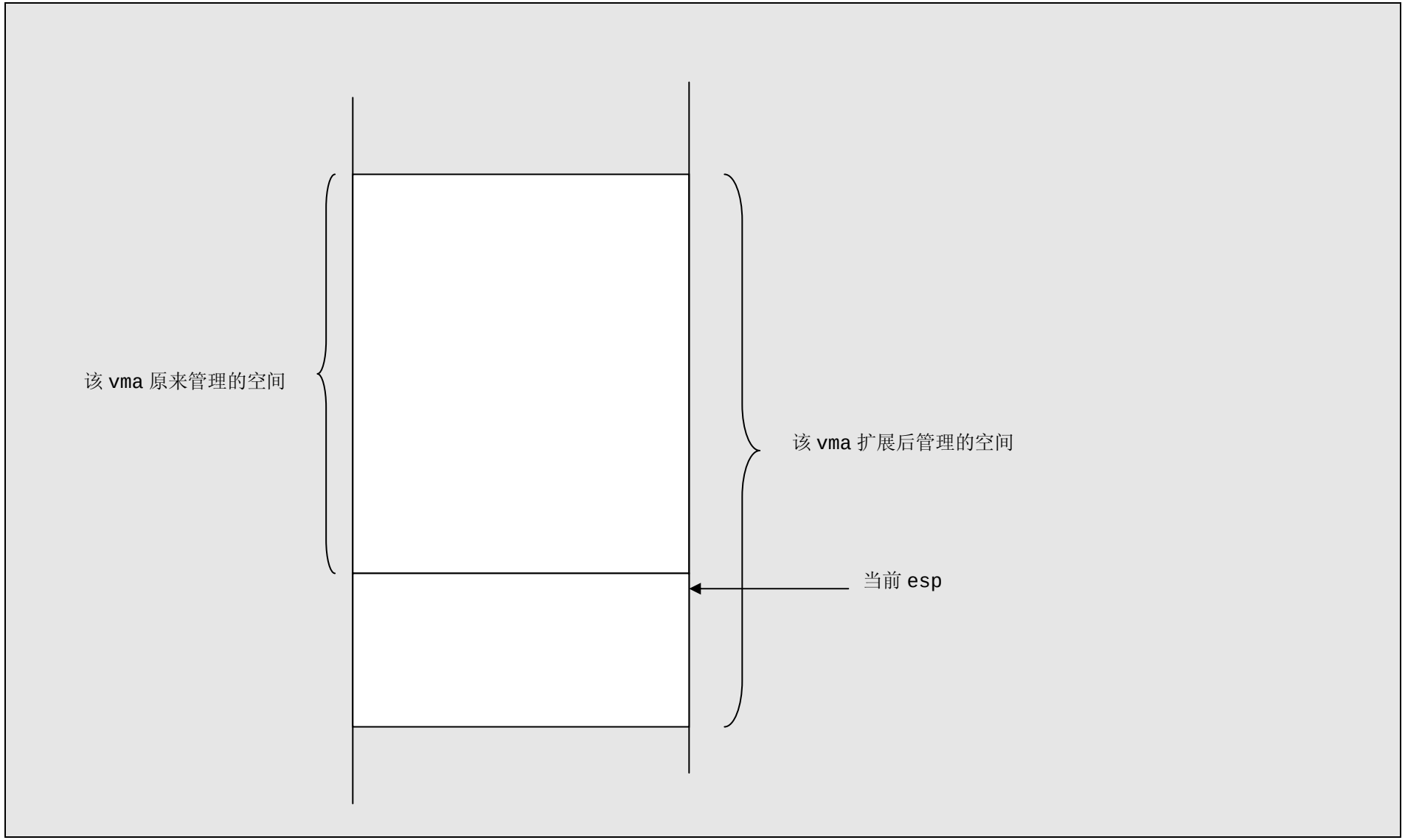
从代码的注释上看，应该是类似下面的情况出现。

本来 esp（stack top register）已经接近于该 vma 管理的起始地址(vma->vm_start)，而这时如果运行 pusha(d)的 opcode，则 esp 将超出 vma->vm_start 的边界，同时由于该指令要压入几乎所有通用寄存器，所以 esp 预留的空间比较大，但最大不超过 32 个 bytes。


管理 stack 的 vma
当前 esp，已经非常接近于 vma->vm_start
pusha 指令会 push 几乎所有 general register，但最多不应超过 32 bytes 的空间。
引发该 page fault 的指令如 pusha(d)等，触发前的 esp 与触发后的 esp 之差不应该超过 32 bytes。

```
    }
    if (expand_stack(vma, address))
        goto bad_area;
```

扩展 stack。即用户态的 stack 是动态扩展的。


该 vma 原来管理的空间
该 vma 扩展后管理的空间
当前 esp

我这里有疑问的是上面的

```
if (error_code & 4)
```

好象是多余的判断。

```
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:
```

运行到这里，表明产生 page fault 的地址最起码在用户空间是存在的。

```
    info.si_code = SEGV_ACCERR;
    write = 0;
    switch (error_code & 3) {
```

error_code 的最低两位的意义如下：

1. error-code = 0，bit 0 与 bit 1 清位，“读”“不存在的页”
2. error-code = 1，bit 0 置位，bit 1 清位，“读”“不能读的页”，在 x86 CPU 上好像没有这种保护。
3. error-code = 2，bit 0 清位，bit 1 置位，“写” “不存在的页”
4. error-code = 3，bit 0 与 bit 1 置位，表示触发该 page fault 的是“写”了“没有写允许”的页面。

上面除了 error_code = 1 是非法状态以外，其他都可以理解。

```
       default:   /* 3: write, present */
#ifdef TEST_VERIFY_AREA
           if (regs->cs == KERNEL_CS)
               printk("WP fault at %08lx\n", regs->eip);
```

```
#endif
          /* fall through */
       case 2:       /* write, not present */
          if (!(vma->vm_flags & VM_WRITE))
              goto bad_area;
```
如果该内存区域是写保护的，自然由“写”出发的 page fault 是非法的。
```
          write++;   设置“写”标志
          break;
       case 1:       /* read, present */
          goto bad_area;
```
这时非法状况。
```

       case 0:       /* read, not present */
          if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
              goto bad_area;
```
要读的该内存区域必须有 VM_READ 或 VM_EXEC。
```
   }

 survive:
   /*
    * If for any reason at all we couldn't handle the fault,
    * make sure we exit gracefully rather than endlessly redo
    * the fault.
    */
```
真正处理 page fault 的函数。address 是产生 page fault 的地址，而 write = 0，表示是“读”操作触发的 page fault，write = 1，表示是“写”操作触发的 page fault。

```
    switch (handle_mm_fault(mm, vma, address, write)) {
```
handle_mm_fault()的返回值实际上就是handle_pte_fault()的返回值。
```
    case 1:
        tsk->min_flt++;
```
min_flt成员变量用于追踪该process产生page fault，但并没有为该page fault载入page的数目。
```
        break;
    case 2:
        tsk->maj_flt++;
```
maj_flt成员变量用于追踪该process产生page fault，并为该page fault载入了page的数目。
```
        break;
    case 0:
```
出错了。
```
        goto do_sigbus;
    default:
        goto out_of_memory;
```
申请不到空闲的page frame，说明内存极度紧张。
```
    }

    /*
     * Did it hit the DOS screen memory VA from vm86 mode?
     */
    if (regs->eflags & VM_MASK) {
        unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
        if (bit < 32)
            tsk->thread.screen_bitmap |= 1 << bit;
    }
```

```
    up_read(&mm->mmap_sem);
    return;
```
上面的代码是处理 x86 架构 PC 的，忽略。

```
/*
 * Something tried to access memory that isn't in our memory map..
 * Fix it, but check if it's kernel or user first..
 */
bad_area:
```
非法地址的处理。
```
    up_read(&mm->mmap_sem);
```
释放占有的 semaphore。

```
    /* User mode accesses just cause a SIGSEGV */
    if (error_code & 4) {
```
如果是用户态触发的 page fault，在 task_struct 中设置相应的信息。

```
        tsk->thread.cr2 = address;
```
记录 page fault address
```
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = 14;
```
page fault 为 14 （0Eh)
```
        info.si_signo = SIGSEGV;
```
通过 signal 来报错，这里信号为 SIGSEGV。

```
        info.si_errno = 0;
```

```
        /* info.si_code has been set above */
        info.si_addr = (void *)address;
        force_sig_info(SIGSEGV, &info, tsk);
```
当该 process 返回到用户态时，首先将执行该 signal handler，用户一般会受到 segment fault。
```
        return;
    }

    /*
     * Pentium F0 0F C7 C8 bug workaround.
     */
    if (boot_cpu_data.f00f_bug) {
        unsigned long nr;

        nr = (address - idt) >> 3;

        if (nr == 6) {
            do_invalid_op(regs, 0);
            return;
        }
    }
```
上面的代码是个 CPU 的补丁，与正常 page fault 处理流程无关。在我的《CPU Bug 与 Linux Kernel》一文中有详述。
```

no_context:
    /* Are we prepared to handle this kernel fault?  */
    if ((fixup = search_exception_table(regs->eip)) != 0) {
        regs->eip = fixup;
```

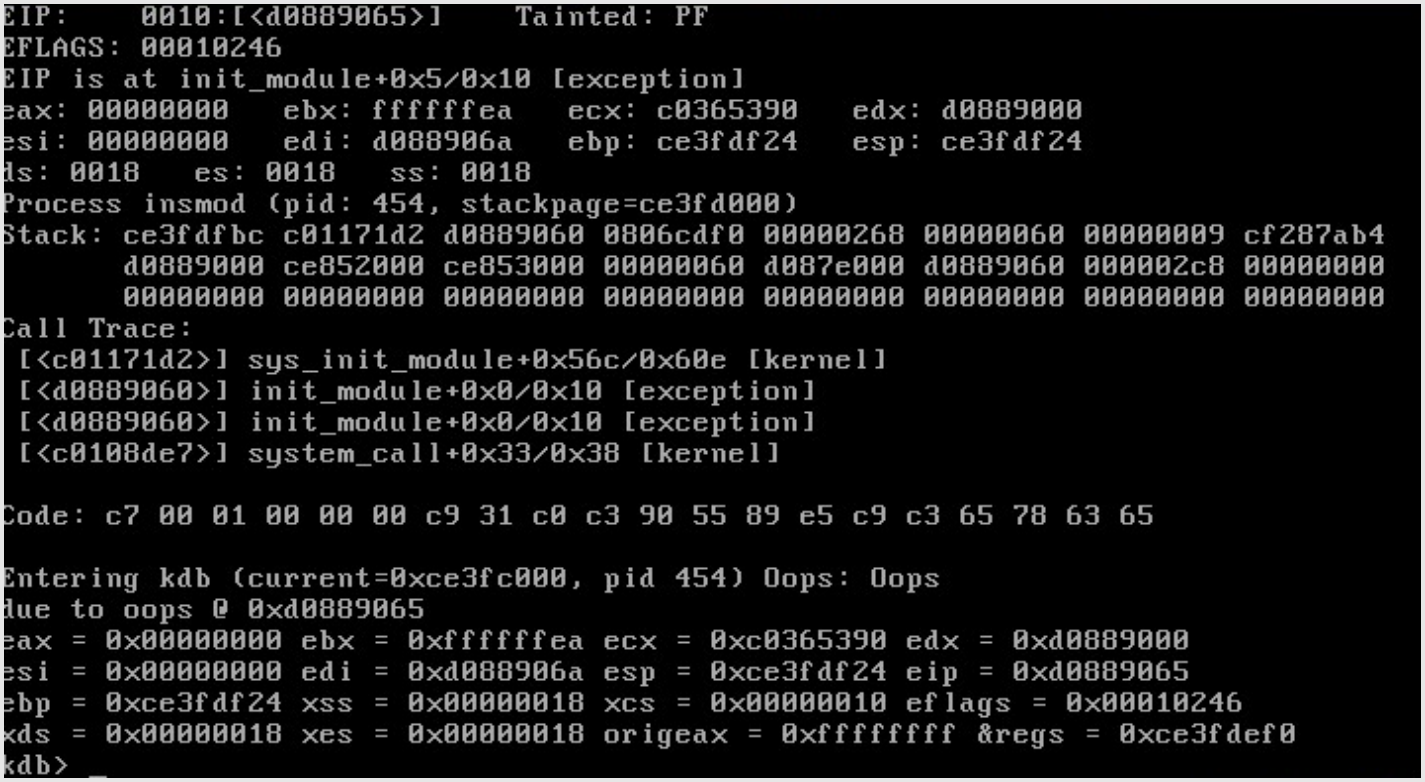
```
        return;
    }
```
对于内核中的 page fault，内核有一定的修补机制。具体分析见我的《Linux 与 SEH》一文。

```
/*
 * Oops. The kernel tried to access some bad page. We'll have to
 * terminate things with extreme prejudice.
 */
```
运行到这里，表示你要倒大霉了。这是个内核级的 bug，只有当机了。

```
    bust_spinlocks(1);

    if (address < PAGE_SIZE)
        printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
    else
        printk(KERN_ALERT "Unable to handle kernel paging request");
    printk(" at virtual address %08lx\n",address);
    printk(" printing eip:\n");
    printk("%08lx\n", regs->eip);
    asm("movl %%cr3,%0":"=r" (page));
    page = ((unsigned long *) __va(page))[address >> 22];
    printk(KERN_ALERT "*pde = %08lx\n", page);
    if (page & 1) {
        page &= PAGE_MASK;
        address &= 0x003ff000;
        page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
```

```
        printk(KERN_ALERT "*pte = %08lx\n", page);
    }
    die("Oops", regs, error_code);
    bust_spinlocks(0);
    do_exit(SIGKILL);
```

上面的代码在干什么呢？是在做临死前的最后交待。你看到的下面的画面就是“最后交待”。



```
/*
 * We ran out of memory, or some other thing happened to us that made
 * us unable to handle the page fault gracefully.
 */
out_of_memory:
```
如果在 handle_mm_fault()中没有找到空闲物理内存来解决本次 page fault，则有些进程要倒霉了。
```
    if (tsk->pid == 1) {
```
对于 init 进程，拿他没办法，总不见的杀 init 进程吧。只能放弃，选下一个倒霉蛋。这里完全看 scheduler 挑哪一个运行了。
```
        yield();
        goto survive;
    }
    up_read(&mm->mmap_sem);
    printk("VM: killing process %s\n", tsk->comm);
    if (error_code & 4)
        do_exit(SIGKILL);
```
对于其他进程就没那么幸运了，随让你在内存极度稀缺的情况下，还要内存，只能杀掉了事。
```
    goto no_context;

do_sigbus:
    up_read(&mm->mmap_sem);

    /*
     * Send a sigbus, regardless of whether we were in kernel
     * or user mode.
     */
    tsk->thread.cr2 = address;
```

```
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo = SIGBUS;
    info.si_errno = 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *)address;
    force_sig_info(SIGBUS, &info, tsk);

    /* Kernel mode? Handle exceptions or die */
    if (!(error_code & 4))
        goto no_context;
    return;
```

通过发 signal 来报错。

```
vmalloc_fault:
```

由于 vmalloc 分配的内存并不是马上兑现的，这里来“兑现”。下面的代码如果你熟悉 x86 CPU 的架构，应该很好理解。

```
    {
        /*
         * Synchronize this task's top level page-table
         * with the 'reference' page table.
         *
         * Do _not_ use "tsk" here. We might be inside
         * an interrupt in the middle of a task switch..
         */
        int offset = __pgd_offset(address);
        pgd_t *pgd, *pgd_k;
```

```
        pmd_t *pmd, *pmd_k;
        pte_t *pte_k;

        asm("movl %%cr3,%0":"=r" (pgd));
        pgd = offset + (pgd_t *)__va(pgd);
        pgd_k = init_mm.pgd + offset;

        if (!pgd_present(*pgd_k))
            goto no_context;
        set_pgd(pgd, *pgd_k);

        pmd = pmd_offset(pgd, address);
        pmd_k = pmd_offset(pgd_k, address);
        if (!pmd_present(*pmd_k))
            goto no_context;
        set_pmd(pmd, *pmd_k);

        pte_k = pte_offset(pmd_k, address);
        if (!pte_present(*pte_k))
            goto no_context;
        return;
    }
}
```

handle_mm_fault 源码解析

```
src/linux-2.4.20/mm/memory.c

vma 管理产生 page fault 的地址 address，write_access = 0 表示触发该 page fault 的是读操作；而 = 1 表示写操作。
/*
 * By the time we get here, we already hold the mm semaphore
 */
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);
由 address 获得 pgd，在 x86 上就是页目录(Page Directory Entry)，即用 address 的高 10 位做为 index，在 CR3 所指示的 table 中取出 PDE。
    /*
     * We need the page table lock to synchronize with kswapd
     * and the SMP-safe atomic PTE updates.
     */
    spin_lock(&mm->page_table_lock);
    pmd = pmd_alloc(mm, pgd, address);
    在 x86 32 位 CPU 上，pmd 是空，等同于 pmd 与 pgd 是重合的。
```

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

```
/*
 * These routines also need to handle stuff like marking pages dirty
 * and/or accessed for architectures that don't do it in hardware (most
 * RISC architectures).  The early dirtying is also good on the i386.
 *
 * There is also a hook called "update_mmu_cache()" that architectures
 * with external mmu caches can use to update those (ie the Sparc or
 * PowerPC hashed page tables that act as extended TLBs).
 *
 * Note the "page_table_lock". It is to protect against kswapd removing
 * pages from under us. Note that kswapd only ever _removes_ pages, never
 * adds them. As such, once we have noticed that the page is not present,
 * we can drop the lock early.
 *
 * The adding of pages is protected by the MM semaphore (which we hold),
 * so we don't need to worry about a page being suddenly been added into
```

```
 * our VM.
 *
 * We enter with the pagetable spinlock held, we are supposed to
 * release it when done.
 */
```
这里的参数 pte 为用来映射 address 的 Page Table Entry
```
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;
    if (!pte_present(entry)) {
```
定义如下
```
#define pte_present(x)  ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
#define _PAGE_PRESENT   0x001
#define _PAGE_PROTNONE  0x080  /* If not present */
```
检查 pte entry 中的 present 位是否置位，即该 pte 所代表的虚拟地址（address）有没有物理地址对应
而_PAGE_PROTNONE ?
```
       /*
        * If it truly wasn't present, we know that kswapd
        * and the PTE updates will not touch it later. So
        * drop the lock.
        */
       if (pte_none(entry))     检查 pte 是否为空
```

```
            return do_no_page(mm, vma, address, write_access, pte);
```
为空，表示这是一页 anobymous page，只有分配给它一页空页即行。
```
        return do_swap_page(mm, vma, address, pte, entry, write_access);
```
不空，则表示该页的内容已经被交换到 swap 空间中，要从 swap 文件或 swap 设备中读取出来。记录在 swap 文件（设备）哪儿的信息在 pte 中。
```
    }
```

运行到这里，表示虚拟地址是有物理地址映射的。
```
    if (write_access) {
```
是写操作引起的 page fault
```
        if (!pte_write(entry))   检查该页是否可写
            return do_wp_page(mm, vma, address, pte, entry);
```
如果不可写，则显然该页需要 Write-Copy(写时拷贝)。
```
        entry = pte_mkdirty(entry);
```
置该页的 pte 的 dirty 位。
```
    }
```
运行到这里表示是读操作引起的 page fault。有点奇怪，虚拟地址 address 与物理地址有映射关系，对它读操作为什么还会 page fault 呢？
当产生 page fault 时，还没有建立映射关系，但运行到这儿时，却有了，是因为由其他 process 把该页引入了内存，所以下面要重新刷新 TLB。
下面 establish_pte（）的定义如下：

```
/*
 * Establish a new mapping:
 *  - flush the old one
 *  - update the page tables
 *  - inform the TLB about the new one
 *
```

```
 * We hold the mm semaphore for reading and vma->vm_mm->page_table_lock
 */
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);
}
```

我想在单 CPU 上应该不会有这样的情况吧。我的分析如下，不知是否对。

当在访问用户态读 address 而产生 page fault，系统马上陷入内核，从 trap 0eh handler 到运行到这儿，在单 CPU 上这时候进程是无法被抢占的，没有可能为 address 建立映射关系。但在多 CPU 上，则完全可能其他进程（可能是同一个程序）访问到同样的 address(而该 address 又是进程见共享的)，建立映射关系。

```
    entry = pte_mkyoung(entry);
    establish_pte(vma, address, pte, entry);
    spin_unlock(&mm->page_table_lock);
    return 1;
}
```

handle_pte_fault()函数根据不同情况来调用不同的子函数来处理。我这里简单总结一下：

1. 如果产生 page fault 的地址 address 还没有与物理内存建立映射关系，并且映射该 address 的 pte 是空的，则调用 do_no_page()来处理。这种情况比如像可执行文件载入时为该文件中的 segment 与载入内存之间建立的映射关系。
2. 如果产生 page fault 的地址 address 还没有与物理内存建立映射关系，并且映射该 address 的 pte 非空，表示该 address 所对应的物理页已经被交换到了 swap 空间去，这时要通过 do_swap_page()函数来读出磁盘上内容，然后重新建立映射关系。
3. 如果产生 page fault 的地址 address 已经与物理内存建立映射关系,并且是写操作,在管理该 address 的 vma 中是允许“写”操作的,但映射该 address 的 pte 却不允许 “写”，表明这是所谓的 Write-On-Copy(COW)页，要调用 do_wp_page()来处理。

下面依次分析这 3 个关键函数。如果你明白了这三个函数，那你对 Unix，甚至连带着对 Windows，将有更深入的理解。

do_no_page()源码解析

```
/*
 * do_no_page() tries to create a new page mapping. It aggressively
 * tries to share with existing pages, but makes a separate copy if
 * the "write_access" parameter is true in order to avoid the next
 * page fault.
 *
 * As this is called only for pages that do not currently exist, we
 * do not need to flush old virtual caches or the TLB.
 *
 * This is called with the MM semaphore held and the page table
 * spinlock held. Exit with the spinlock released.
 */
```

这里 vma 是管理产生 page fault 的 address

write_access = 0，表示“读”address 产生的 page fault；而 = 1，表示“写”address 产生的 page fault

page_table 是为 address 建立映射关系的结构，依赖于 CPU。

```
static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma,
    unsigned long address, int write_access, pte_t *page_table)
{
    struct page * new_page;
    pte_t entry;
```

```
    if (!vma->vm_ops || !vma->vm_ops->nopage)
        return do_anonymous_page(mm, vma, page_table, write_access, address);
```

如果 vma 中对应的操作函数为空，表示这是匿名页，即只要分配一页给地址 address，至于内容是无所谓的，这里调用 do_anonymous_page()来处理。
如果是象载入可执行文件之类的映射而引起的 page fault，自然是对载入的内容有要求的。

```
    spin_unlock(&mm->page_table_lock);

    new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
```

调用的是该结构中的 nopage()。对载入可执行文件之类应该是 src/linux-2.4.20/mm/filemap.c 中的 filemap_nopage()函数。见下面的分析。
返回的是已经读入正确内容的物理页框（Page Frame）。

```
    if (new_page == NULL)    /* no page was available -- SIGBUS */
        return 0;
    if (new_page == NOPAGE_OOM)
        return -1;

    /*
     * Should we do an early C-O-W break?
     */
    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);
    /*
     * This silly early PAGE_DIRTY setting removes a race
     * due to the bad i386 page protection. But it's valid
     * for other architectures too.
     *
     * Note that if write_access is true, we either now have
     * an exclusive copy of the page, or this is a shared mapping,
     * so we can make it writable and dirty to avoid having to
     * handle that later.
     */
    /* Only go through if we didn't race with anybody else... */
    if (pte_none(*page_table)) {
```

映射 address 的 pte 还保持是空的，正常。

```
        ++mm->rss;
```

本进程多占用了一页物理内存。mm->rss 记录当前进程占有的物理内存页数。

```
        flush_page_to_ram(new_page);
```

这是个与 CPU 体系结构相关的函数，在 x86 CPU 上为空。

```
        flush_icache_page(vma, new_page);
```
在 x86 CPU 上同样为空。

```
        entry = mk_pte(new_page, vma->vm_page_prot);
```
实实在在的建立映射关系。

```
        if (write_access)
            entry = pte_mkwrite(pte_mkdirty(entry));
```
如果是“写”，则使得该页为 dirty。
```
        set_pte(page_table, entry);
    } else {
```
映射 address 的 pte 已经非空。

```
        /* One of our sibling threads was faster, back out. */
```
上面原作者的注释已经说明了。因为上面的读取该页内容的函数

```
new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
```
会挂起本进程。在它挂起期间，该页被其他 thread 调入，所以这里什么都不用做。

```
        page_cache_release(new_page);
```
释放该 page frame。
```
        spin_unlock(&mm->page_table_lock);
        return 1;
    }

    /* no need to invalidate: a not-present page shouldn't be cached */
```

```
    update_mmu_cache(vma, address, entry);
    spin_unlock(&mm->page_table_lock);
    return 2;  /* Major fault */
}
```

filemap_nopage 源码解析

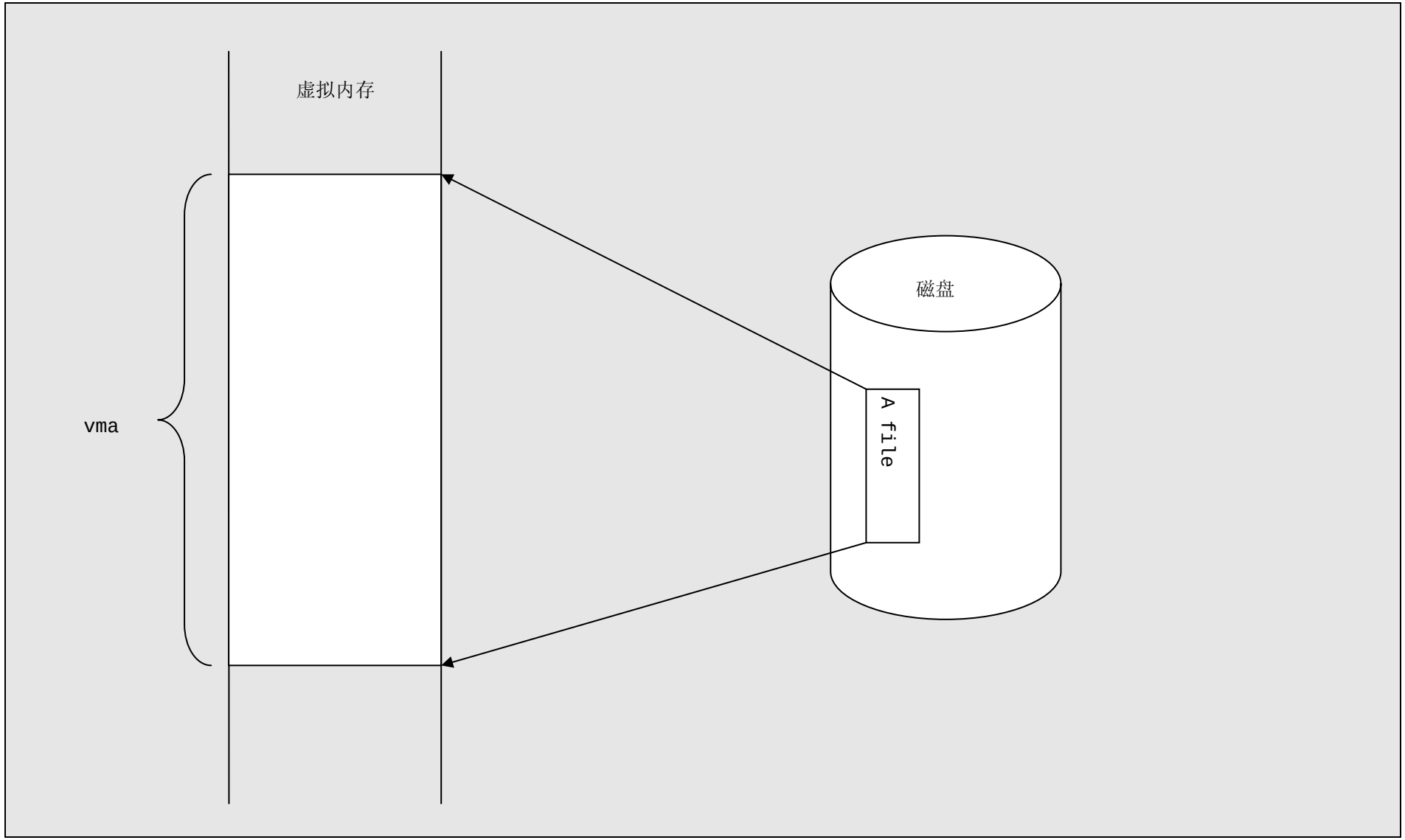
```
src/linux-2.4.20/mm/ filemap_nopage.c

/*
 * filemap_nopage() is invoked via the vma operations vector for a
 * mapped memory region to read in file data during a page fault.
 *
 * The goto's are kind of ugly, but this streamlines the normal case of having
 * it in the page cache, and handles the special cases reasonably without
 * having a lot of duplicated code.
 */
```

area 为管理 page fault 产生地址 address。

本函数就是实现虚拟内存中的一段与某个文件中的一段建立映射。

整个函数因为下面几点而不太易读：

1. 充满 goto，至于为什么用 goto，在函数头上的注释中已经解释了。
2. 由于性能的原因，并不是简单的从文件的某处读出即可。这里先要检查所读内容是否在 cache 中，如果不在，还要做一定的预读。


虚拟内存
vma
磁盘
A file

```
struct page * filemap_nopage(struct vm_area_struct * area, unsigned long address, int unused)
{
    int error;
    struct file *file = area->vm_file;
```
area->vm_file 中为该 vma 映射到的文件。

```
    struct address_space *mapping = file->f_dentry->d_inode->i_mapping;
    struct inode *inode = mapping->host;
    struct page *page, **hash;
    unsigned long size, pgoff, endoff;

    pgoff = ((address - area->vm_start) >> PAGE_CACHE_SHIFT) + area->vm_pgoff;
```
这里 area->vm_pgoff 为文件内的偏移（已经在页边界上对齐）
pgoff 为产生 page fault 的地址 address 在影射的文件内的以页为单位的偏移。

```
    endoff = ((area->vm_end - area->vm_start) >> PAGE_CACHE_SHIFT) + area->vm_pgoff;
```
endoff 为该 vma 的结束地址在射的文件内的以页为单位的偏移。

```
retry_all:
    /*
     * An external ptracer can access pages that normally aren't
     * accessible..
     */
    size = (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
```
inode->i_size 为被映射的文件大小。
size 为已经对齐在页边界上的文件大小。

```
    if ((pgoff >= size) && (area->vm_mm == current->mm))
        return NULL;
```

pgoff >= size 表示产生 page fault 的地址对应的在文件中的位置超过该文件大小

area->vm_mm == current->mm 表示产生 page fault 的虚拟地址空间不是当前的虚拟地址空间。这一条不是太明白。

```
    /* The "size" of the file, as far as mmap is concerned, isn't bigger than the mapping */
    if (size > endoff)
        size = endoff;

    /*
     * Do we have something in the page cache already?
     */
    hash = page_hash(mapping, pgoff);
```

计算 hash key。

```
retry_find:
    page = __find_get_page(mapping, pgoff, hash);
    if (!page)
        goto no_cached_page;
```

从 cache 中查找该页。如果返回空，表示不再 cache 中。

```
    /*
     * Ok, found a page in the page cache, now we need to check
     * that it's up-to-date.
     */
```

在 page cache 中，检查该页是否反映当前状态。

```
    if (!Page_Uptodate(page))
```

```
        goto page_not_uptodate;
```
Page_Uptodate 是个 macro，其定义如下：
```
#define Page_Uptodate(page) test_bit(PG_uptodate, &(page)->flags)
```
测试 PG_uptodate 位是否置位。如果置位才表示该 page 中包含有效数据，具体分析属于 Linux kernel 中的“Page Cache”模块。

下面是如果该 cache 中的 page 包括有效数据
```
success:
    /*
     * Try read-ahead for sequential areas.
     */
    if (VM_SequentialReadHint(area))
        nopage_sequential_readahead(area, pgoff, size);
```
上面是如果置了预读位的话，则预读。
```
    /*
     * Found the page and have a reference on it, need to check sharing
     * and possibly copy it over to another page..
     */
    mark_page_accessed(page);
    flush_page_to_ram(page);
```
在 x86 CPU 上该函数为空。
```
    return page;
```
返回找到的该 page frame。
```
no_cached_page:
```
在 page cache 中没找到只有从文件读取了。

```
    /*
     * If the requested offset is within our file, try to read a whole
     * cluster of pages at once.
     *
     * Otherwise, we're off the end of a privately mapped file,
     * so we need to map a zero page.
     */
    if ((pgoff < size) && !VM_RandomReadHint(area))
        error = read_cluster_nonblocking(file, pgoff, size);
    else
        error = page_cache_read(file, pgoff);
```

上面是具体去读取了。这里是与 Page Cache 打交道了。

```
    /*
     * The page we want has now been added to the page cache.
     * In the unlikely event that someone removed it in the
     * meantime, we'll just come back here and read it again.
     */
    if (error >= 0)
        goto retry_find;

    /*
     * An error return from page_cache_read can result if the
     * system is low on memory, or a problem occurs while trying
     * to schedule I/O.
     */
```

```
    if (error == -ENOMEM)
        return NOPAGE_OOM;
```
内存不够，返回失败。

```
    return NULL;

page_not_uptodate:
```
如果从 cache 中取出的 page frame 中内容已经无效
```
    lock_page(page);
```
先锁住该页。这里的“锁住”，是指占有访问该 page frame 的“锁”，以免发生竞态。

```
    /* Did it get unhashed while we waited for it? */
    if (!page->mapping) {
        UnlockPage(page);
        page_cache_release(page);
        goto retry_all;
    }

    /* Did somebody else get it up-to-date? */
    if (Page_Uptodate(page)) {
        UnlockPage(page);
        goto success;
    }
```
在上面“lock_page(page)”可能 wait，因为有其他执行线程或中断等正在锁住该页，并使该页的内容有效。所以在此再次检查是否有效，如果有效，则解锁该页，然后成功返回。

```
    if (!mapping->a_ops->readpage(file, page)) {
```
如果 page frame 中内容还未有效，则从文件中读出该页。
```
        wait_on_page(page);
        if (Page_Uptodate(page))
            goto success;
    }
    /*
     * Umm, take care of errors if the page isn't up-to-date.
     * Try to re-read it _once_. We do this synchronously,
     * because there really aren't any performance issues here
     * and we need to check for errors.
     */
```
读取失败的情况。
```

    lock_page(page);

    /* Somebody truncated the page on us? */
    if (!page->mapping) {
        UnlockPage(page);
        page_cache_release(page);
        goto retry_all;
    }

    /* Somebody else successfully read it in? */
    if (Page_Uptodate(page)) {
        UnlockPage(page);
```

```
        goto success;
    }
    ClearPageError(page);
    if (!mapping->a_ops->readpage(file, page)) {
        wait_on_page(page);
        if (Page_Uptodate(page))
            goto success;
    }

    /*
     * Things didn't work out. Return zero to tell the
     * mm layer so, possibly freeing the page cache page first.
     */
    page_cache_release(page);
    return NULL;
}
```

do_swap_page 源码解析

```
src/linux-2.4.20/mm/memory.c

/*
 * We hold the mm semaphore and the page_table_lock on entry and
 * should release the pagetable lock on exit..
 */
```

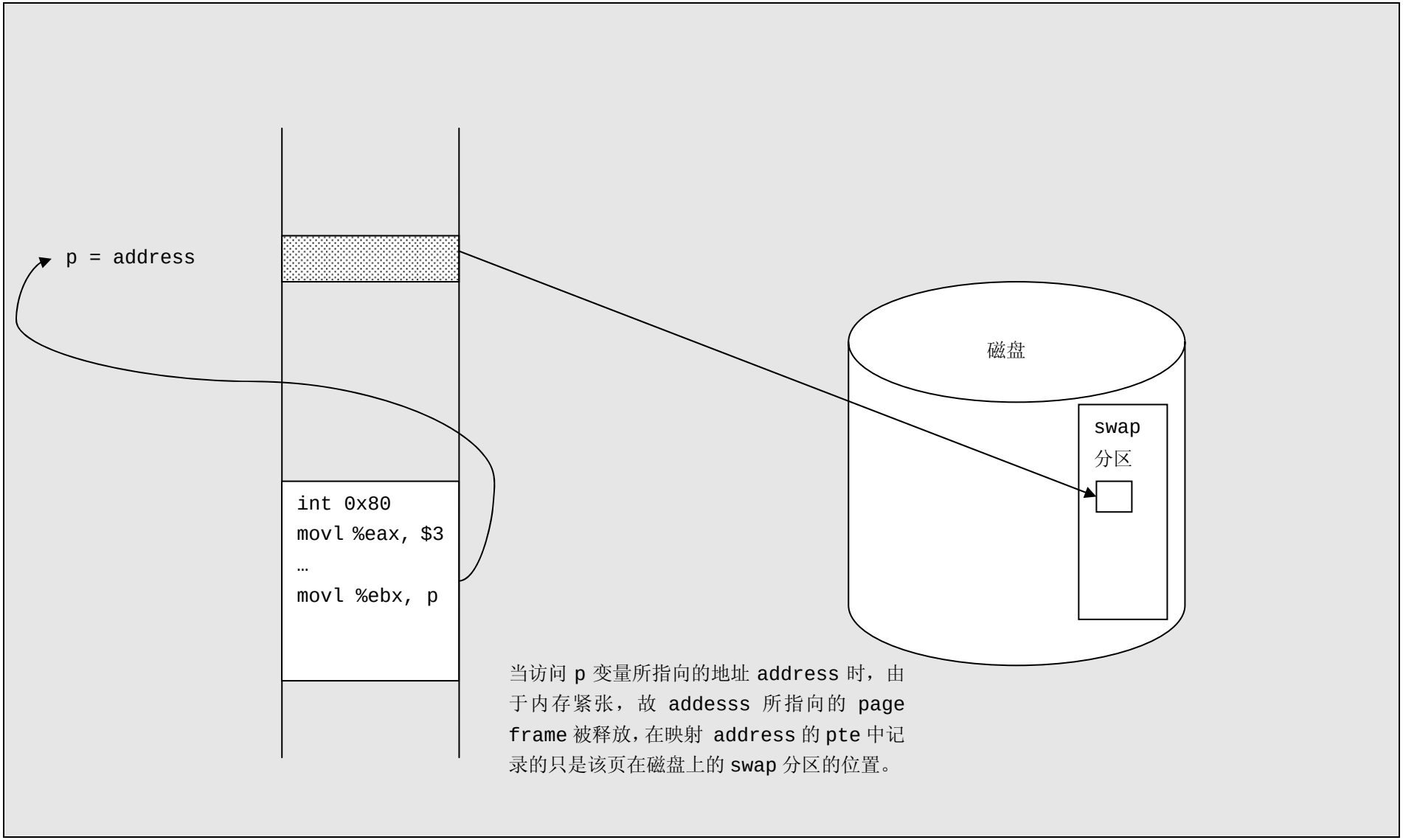
该函数从 swap 设备或文件中取出该页的内容，并为 address 建立映射关系。

vma 管理产生 page fault 地址的 address。

page_table 为指向映射 address 的 pte 结构。

orig_pte 为 page_table 中的内容，这里记录着该页在 swap 空间中的位置。

write_access = 0 表示读操作触发 page fault；而 = 1 表示写操作触发 page fault。



当访问 p 变量所指向的地址 address 时，由于内存紧张，故 addesss 所指向的 page frame 被释放，在映射 address 的 pte 中记录的只是该页在磁盘上的 swap 分区的位置。

```
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)
{
    struct page *page;
    swp_entry_t entry = pte_to_swp_entry(orig_pte);
```
对 swap 出去的页而言，这时的 pte 记录着 swap entry。
```
    pte_t pte;
    int ret = 1;

    spin_unlock(&mm->page_table_lock);
    page = lookup_swap_cache(entry);
```
在 swap 的 cache 中寻找该页。
```
    if (!page) {
```
不在 cache 中。

下面是安排读取磁盘上的 swap page。出于性能的考虑，要预读。
```
        swapin_readahead(entry);
        page = read_swap_cache_async(entry);
        if (!page) {
            /*
             * Back out if somebody else faulted in this pte while
             * we released the page table lock.
             */
            int retval;
            spin_lock(&mm->page_table_lock);
            retval = pte_same(*page_table, orig_pte) ? -1 : 1;
```

```
            spin_unlock(&mm->page_table_lock);
            return retval;
        }

        /* Had to read the page from swap area: Major fault */
        ret = 2;
    }
```
在 cache 中找到。

```
    mark_page_accessed(page);

    lock_page(page);

    /*
     * Back out if somebody else faulted in this pte while we
     * released the page table lock.
     */
    spin_lock(&mm->page_table_lock);
    if (!pte_same(*page_table, orig_pte)) {
```
如果不相等，说明该页已经被引入，并且虚拟地址与物理 page frame 的映射关系也已经建立好。这时，orig_pte 中是原来记录的该虚拟页在 swap 分区上的位置，而*page_table 中则是 address 到物理 page frame 的真正映射关系，所以不相等。

```
        spin_unlock(&mm->page_table_lock);
        unlock_page(page);
        page_cache_release(page);
        return 1;
```
释放 cache 中该页，因为该页已经不在磁盘的 swap 分区上。

```
    }

    /* The page isn't present yet, go ahead with the fault. */

    swap_free(entry);
```
该页被引入内存，自然磁盘上的“页”可以释放了。

```
    if (vm_swap_full())
        remove_exclusive_swap_page(page);

    mm->rss++;
```
统计本进程占有物理内存计数加 1。

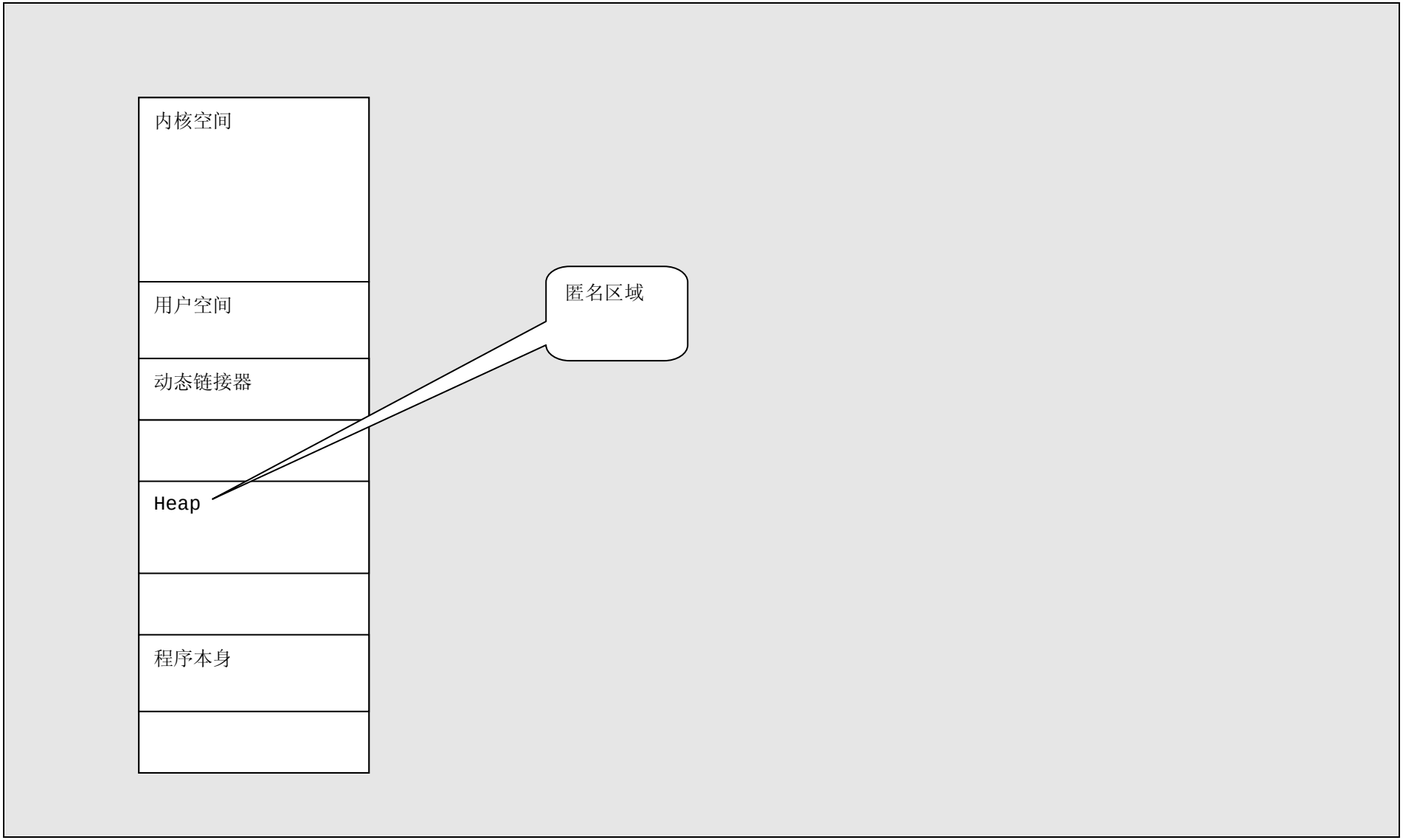
```
    pte = mk_pte(page, vma->vm_page_prot);
```
生成映射关系的 pte 项。
```
    if (write_access && can_share_swap_page(page))
        pte = pte_mkdirty(pte_mkwrite(pte));
```
如果是“写”操作，则置可写与脏属性。
```
    unlock_page(page);

    flush_page_to_ram(page);
    flush_icache_page(vma, page);
    set_pte(page_table, pte);
```
建立映射关系。

```
    /* No need to invalidate - it was non-present before */
```

```
    update_mmu_cache(vma, address, pte);
    spin_unlock(&mm->page_table_lock);
    return ret;
}
```

do_anonymous_page 源码解析

```
src/linux-2.4.20/mm/memory.c
```


内核空间
用户空间
动态链接器
Heap
程序本身
匿名区域

所谓匿名（anonymous）区域就是虚拟地址空间中没有与任何文件有映射关系的区域。比如常规的 heap 区即是。
vma 管理的匿名区域
addr 为产生在匿名区域的 page fault 地址
page_table 为映射 addr 地址的 pte 指针
write_access = 0，即读； = 1，即写

由于匿名区域不与任何磁盘上文件建立关系，所以处理也比较简单。如果是对匿名区域的“读”，则把系统中固定的“zero page”给用户，用户读出的都是被初始化为零的数据。如果是“写”，则要分配一页 page frame，然后把它与 addr 建立映射关系。

```
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;

    /* Read-only mapping of ZERO_PAGE. */
    entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
```

这里的 ZERO_PAGE(addr)定义如下：

```
/*
 * ZERO_PAGE is a global shared page that is always zero: used
 * for zero-mapped memory areas etc..
 */
extern unsigned long empty_zero_page[1024];
#define ZERO_PAGE(vaddr) (virt_to_page(empty_zero_page))
```

即把上面系统中的 empty_zero_page[1024]映射到虚拟地址 addr，并且是“write-protected”。由此可见内核对“读”匿名页引起的 page fault，并不真正分配内存，而是把固定的所谓“zero page”返回给用户。只有当程序去写该 page 时，由于该页的属性是“write-protected”的，所以会再次出发 page fault，但这次是会运行 do_wp_page（）函数。在那个函数里，内核会真正分配一页，并允许“write”。形象一点，下面的代码会触发两次 page fault。

```
int nTest = *pInt;      ①
*pInt = 3;              ②
```
假设 pInt 所指向的空间是匿名区域的话，则①会触发调用 do_anonymous_page（）函数，而②则会触发调用 do_wp_page（）函数。

```
   /* ..except if it's a write access */
   if (write_access) {
```
如果是“写”引起的 page fault

```
       struct page *page;

       /* Allocate our own private page. */
       spin_unlock(&mm->page_table_lock);

       page = alloc_page(GFP_HIGHUSER);
       if (!page)
           goto no_mem;
```
分配一页 page frame。

```
       clear_user_highpage(page, addr);
```
把该页的内容清零。

```
       spin_lock(&mm->page_table_lock);
       if (!pte_none(*page_table)) {
```
表示有其他执行路径把当前 page fault 的 page 已经载入（因为上面的获取 mm-> page_table_lock 可能挂起本进程）

```
           page_cache_release(page);
```

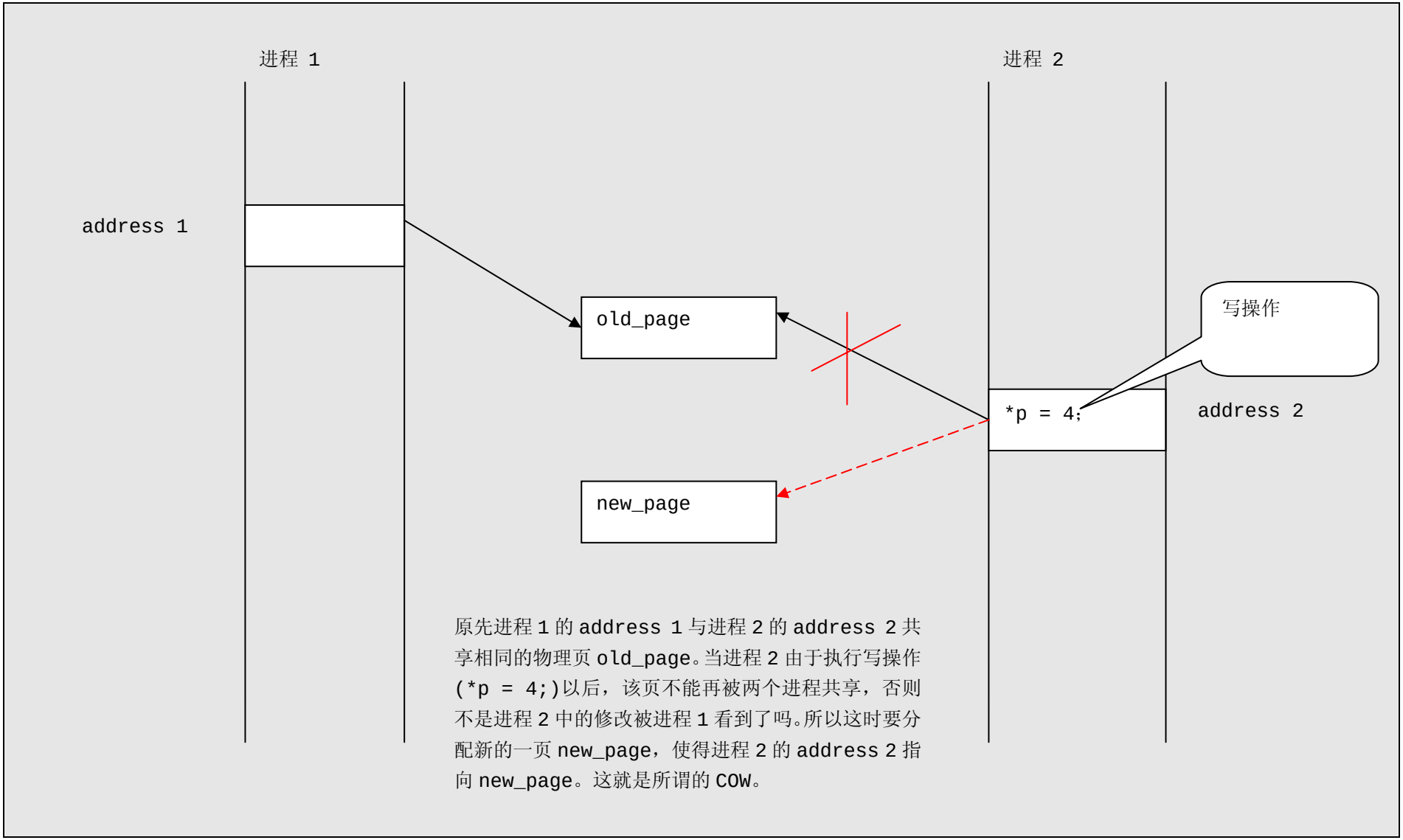
```
            spin_unlock(&mm->page_table_lock);
            return 1;      返回 1，表示本进程的 page fault 产生了，但并没有载入一页 page frame
        }
```
运行到这里表示没有其他执行路径载入该页。
```
        mm->rss++;     该 process 多占有了一页物理内存
        flush_page_to_ram(page);
```
在 x86 CPU 上为空。
```

        entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
```
根据物理地址与该 vma 中的区域属性建立 pte。这里因为是“写”操作，所以置 dirty 位，并且该页“可写”。
```

        lru_cache_add(page);
        mark_page_accessed(page);
    }

    set_pte(page_table, entry);
```
建立映射。
```

    /* No need to invalidate - it was non-present before */
    update_mmu_cache(vma, addr, entry);
    spin_unlock(&mm->page_table_lock);
    return 1;  /* Minor fault */

no_mem:
    return -1;
}
```

do_wp_page 源码解析

```
src/linux-2.4.20/mm/memory.c

该 handler 用于处理 Copy-On-Write(COW)的 page,也就是对原本共享的页，当有 process 对其进行改写时，该共享页必须被“私有化”。

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * Goto-purists beware: the only reason for goto's here is that it results
 * in better assembly code.. The "default" path will see no jumps at all.
 *
 * Note that this routine assumes that the protection checks have been
 * done by the caller (the low-level page fault routine in most cases).
 * Thus we can safely just mark it writable once we've done any necessary
 * COW.
 *
 * We also mark the page dirty at this point even though the page will
 * change only once the write actually happens. This avoids a few races,
 * and potentially makes it more efficient.
 *
 * We hold the mm semaphore and the page_table_lock on entry and exit
```

```
 * with the page_table_lock released.
 */
```

vma 管理着包含地址 address 的共享区域，而 page_table 为指向映射 address 与物理页的 pte 的指针，pte 即是映射关系的 entry。



原先进程 1 的 address 1 与进程 2 的 address 2 共享相同的物理页 old_page。当进程 2 由于执行写操作(*p = 4;)以后，该页不能再被两个进程共享，否则不是进程 2 中的修改被进程 1 看到了吗。所以这时要分配新的一页 new_page，使得进程 2 的 address 2 指向 new_page。这就是所谓的 COW。

```
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct * vma,
    unsigned long address, pte_t *page_table, pte_t pte)
{
    struct page *old_page, *new_page;

    old_page = pte_page(pte);
```

取得原共享页的page frame。

```
    if (!VALID_PAGE(old_page))
        goto bad_wp_page;
```

如果该页无效，自然报错退出。

```
    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_mkdirty(pte_mkwrite(pte))));
            spin_unlock(&mm->page_table_lock);
            return 1;  /* Minor fault */
        }
    }

    /*
     * Ok, we need to copy. Oh, well..
     */
    page_cache_get(old_page);
```

增加原有 page（被共享的页）的引用计数。

```
    spin_unlock(&mm->page_table_lock);

    new_page = alloc_page(GFP_HIGHUSER);
    if (!new_page)
        goto no_mem;
```

分配一个 Page Frame。

```
    copy_cow_page(old_page,new_page,address);
```

这个函数是把在 old_page 中的内容拷贝到 new_page 中。代码如下：

```
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);
}
```

该函数中的条件判断是针对 do_anonymous_page（）函数的。在那个函数中，如果是读取匿名页而引起的 page fault，则内核并不是分配一页，初始化成零以后再返回给用户；而是把固定的所谓 zero page 返回。这样的好处是如果在程序中只是读取，没有“写”的话，那系统中所有的匿名页实际上只占用一页物理页。但当程序要写该页时，则不能再共享了，必须实打实分配了。

```
    /*
     * Re-check the pte - we dropped the lock
     */
```

```
    spin_lock(&mm->page_table_lock);
    if (pte_same(*page_table, pte)) {
```

由于上面分配页框的函数 alloc_page(GFP_HIGHUSER)可能会挂起该进程（在内存暂时不能满足的情况下），可能在本进程挂起期间，由其他进程把 address 所代表的页面引入了内存。这里就是判断这种情况。如果 same，表示该种情况成立。

```
        if (PageReserved(old_page))
            ++mm->rss;
        break_cow(vma, new_page, address, page_table);
        lru_cache_add(new_page);

        /* Free the old page.. */
        new_page = old_page;
    }
    spin_unlock(&mm->page_table_lock);
    page_cache_release(new_page);
    page_cache_release(old_page);
    return 1;  /* Minor fault */

bad_wp_page:
    spin_unlock(&mm->page_table_lock);
    printk("do_wp_page: bogus page at address %08lx (page 0x%lx)\n",address,(unsigned long)old_page);
    return -1;
no_mem:
    page_cache_release(old_page);
    return -1;
}
```

结论

本文整理自 3 年多前的“读核”笔记，所以是 2.4 的内核。从 2.4 到 2.6，文件映射本身没有太大改变，倒是文件映射中处处依赖的磁盘 cache 的实现变化非常大。文件映射是蛮有趣的一块，因为它是内存与文件的两者胶合的地方，所以涉及的交互的模块也较多，错综复杂。读代码时，很容易迷失。我直到现在还没彻底理清呢！^_^

如有指教，请一定相告。

联系

Walter Zhou
z-l-dragon@hotmail.com