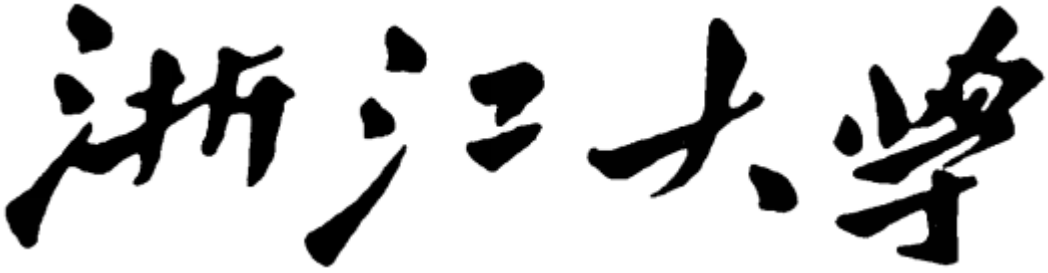


研究报告

课程名称:	**操作系统**
姓 名:	臧可
学 院:	计算机科学与技术学院
系:	计算机科学与技术系
专 业:	计算机科学与技术学院
学 号:	3180102095
指导教师:	夏莹杰

2020年　12月　28日

Linux内存管理

一、研究问题

1. 分析Linux的do_page_fault()函数

二、内核版本

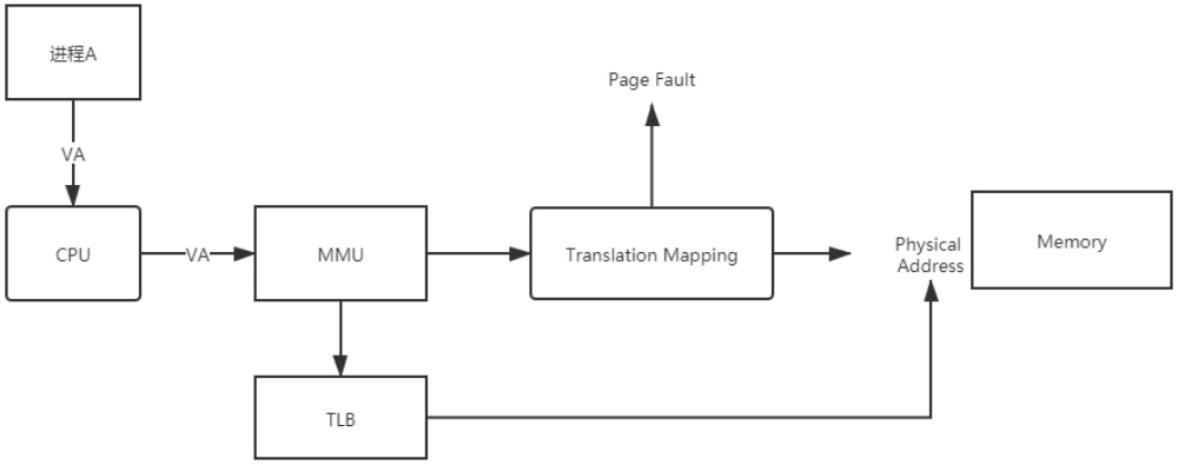
linux-4.20.1

三、研究分析

3.1 缺页异常基本原理

缺页错误，指的是硬件错误、硬中断、分页错误、寻页缺失、缺页中断、页故障等，当软件试图访问已映射在虚拟地址空间中，但目前并未加载在物理内存中的一个分页时，由中央处理器的内存管理单元所发出的中断。

CPU通过地址总线可以访问连接在地址总线上的所有外设，包括物理内存、IO设备等等，但从CPU发出的访问地址并非是这些外设在地址总线上的物理地址，而是一个虚拟地址，由MMU将虚拟地址转换成物理地址再从地址总线上发出，MMU上的这种虚拟地址和物理地址的转换关系是需要创建的，并且MMU还可以设置这个物理页是否可以进行写操作，当没有创建一个虚拟地址到物理地址的映射，或者创建了这样的映射，但那个物理页不可写的时候，MMU将会通知CPU产生了一个缺页异常。



缺页错误的分类：

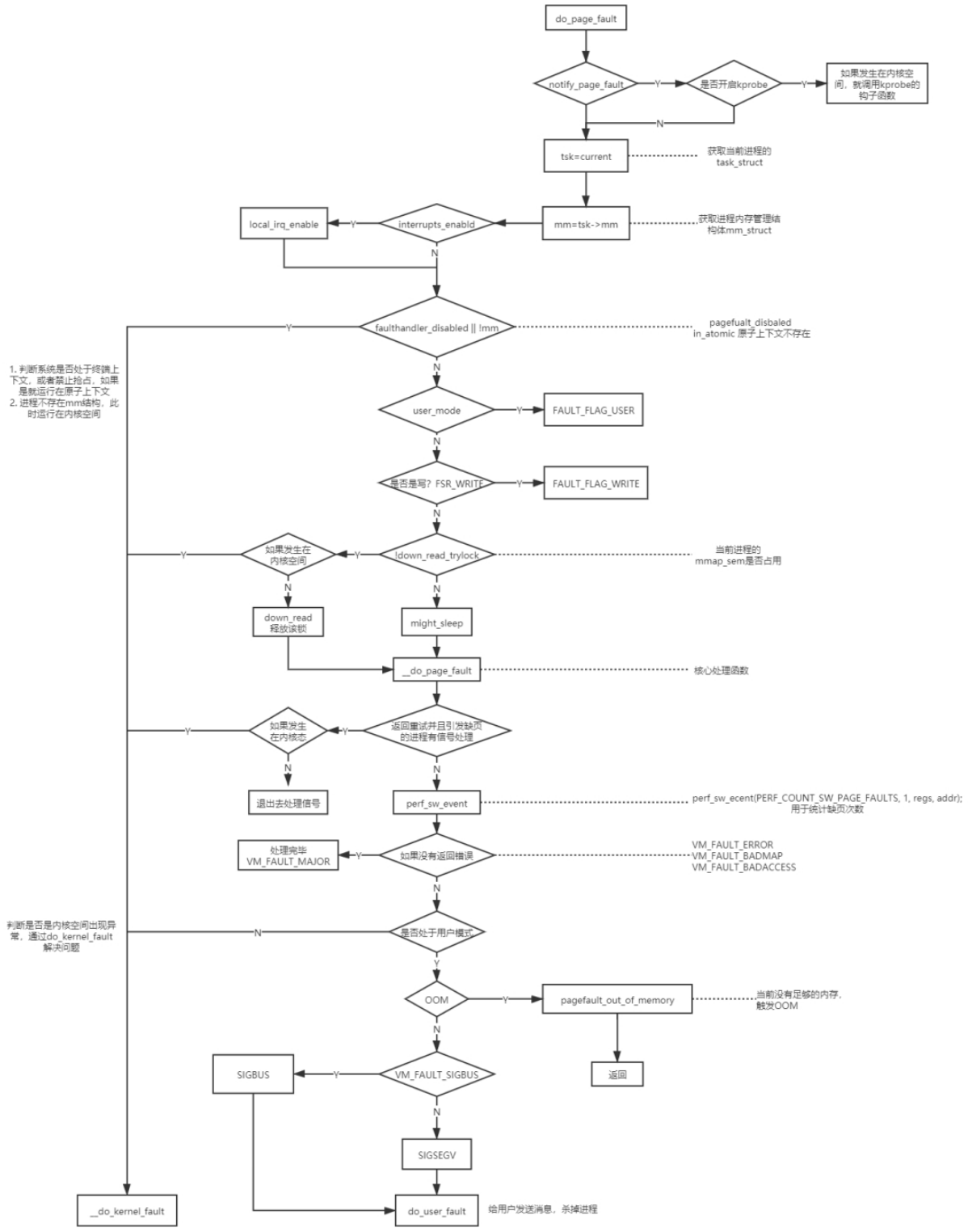
- **硬件缺页(Hard Page Fault):** 此时物理内存中没有对应的页帧，需要CPU打开磁盘设备读取到物理内存中，再让MMU建立VA和PA的映射
- **软缺页(Soft Page Fault):** 此时物理内存中存在对应的页帧，只不过可能是其他进程调入，发生缺页异常的进程不知道，此时MMU只需要重新建立映射即可，无需从磁盘写入内存，一般出现在多进程共享内存区域
- **无效缺页(Invalid Page Falut):** 比如进程访问的内存地址越界访问，空指针引用就会报段错误等

常见的场景：

- **地址空间映射关系未建立**：
  - 内核提供了很多申请内存的接口函数malloc/mmap，申请的虚拟地址空间，但是并未分配实际的物理页面，当首次访问的时候将会触发缺页异常
  - 用户态的经常要进行地址访问，在进程刚创建运行时，页会伴随着大量的缺页异常，例如文件页(代码段/数据段)映射到进程地址空间，首次访问会产生缺页异常
- **地址空间映射已建立**：
  - 当访问的页面已经被swapping到磁盘，访问时触发缺页异常
  - fork子进程时，子进程共享父进程的地址空间，写是触发缺页异常(COW技术)
  - 要访问的页面被KSM合并，写时触发缺页异常(COW技术)
- **访问的地址空间不合法：**
  - 用户空间访问内核空间地址，触发缺页异常
  - 内核空间访问用户空间地址，触发缺页异常

3.2 do_page_fault原理

对于页表错误(即线性地址无效，没有对应的物理地址)和页权限错误，都会调用到缺页异常的核心函数do_page_fault，该函数是和体系结构相关的一个函数。缺页异常的来源可分为两种，一种是内核空间(访问了线性地址空间的第4个GB)，一种是用户空间(访问了线性地址空间的0~3GB)



核心函数do_page_fault，其处理流程如上图所示，简述其处理过程：

- in_atomic判断当前状态是否处于中断上下文或者禁止抢占状态，如果是，说明运行在原子上下文中，那么就跳转到内核处理异常接口do_kernel_fault；同时如果当前进程没有struct mm_struct结构，说明这是一个内核线程，同样进入到do_kernel_fault中
- 如果是用户模式，那么flags置位FAULT_FLAG_USER
- down_read_trylock判断当前的进程的mm->mmap_sem读写信号量释放可以获取，返回1是表示成功获取，返回0则表示被人占用，被人占用时分为两种情况
    - 一种发生在内核空间，如果没有在exception_tables中查询到该地址，就跳转到do_kernel_fault

    - 一种发生在用户空间，可以调用down_read来睡眠等待持有该锁的所有者释放该锁
- 通过__do_page_fault来完成查找合适的vma，并分配，并返回分配后的状态
- 如果没有返回错误类型，说明缺页中断处理完成，其他异常分为
    - 如果返回错误，且当前处于内核模式，那么就跳转到__do_kernel_fault
    - 如果错误类型是VM_FAULT_OOM，说明当前系统中没有足够的内存，那么就调用pagefault_out_of_memory函数来触发OOM机制
    - 如果是VM_FAULT_SIGBUS，就调用__do_user_fault向用户空间发送SIGBUS信号，杀死进程，其他的错误则发送SIGSEGV的段错误，杀死进程
- 如果错误发生在内核模式，如果内核无法处理，那么只能调用__do_kernel_fault来处理

3.3 源码分析

3.3.1 do_page_fault()

```
dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    /* CR2寄存器中包含有最新的页错误发生时的虚拟地址 */
    unsigned long address = read_cr2(); /* Get the faulting address */
    /*
      ...
    */
    __do_page_fault(regs, error_code, address); /* 处理缺页中断 */
        exception_exit(prev_state);
}
NOKPROBE_SYMBOL(do_page_fault);
```

do_page_fault()函数入口。regs是struct pt_regs结构的指针，保存了在发生异常的寄存器内容。error_code是一个32位长整型数据，但是只有最低3位有效，在异常发生时，由CPU的控制部分根据系统当前上下文的情况，生成此3位数据，压入堆栈。

这3位的含义表示：

	set(=1)	clear(=0)
0位s(1b)	保护性错误，越权访问产生异常	“存在位”为0，要访问的页面不在RAM中导致异常
1位(10b)	因为写访问导致异常(write)	因为读或者运行产生异常(read or execute)
2位(100b)	用户态(User Mode)	内核态(Kernel Mode)

宏定义read_cr2()是一组汇编指令。在发生缺页异常时，CPU会将发生缺页异常的地址拷贝到cr2控制寄存器中，然后进入缺页异常的处理过程。这段汇编指令以及宏调用，将该地址从cr2中取出，然后存在在address变量中。

3.3.2 __do_page_fault()

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
                unsigned long address)
```

```
{
        prefetchw(&current->mm->mmap_sem);

        if (unlikely(kmmio_fault(regs, address)))
                return;

        /*
     * Was the fault on kernel-controlled part of the address space?
     * 检查address来判断地址属于内核态还是用户态
     */
        if (unlikely(fault_in_kernel_space(address)))
        /* 处理内核态的缺页中断 */
                do_kern_addr_fault(regs, hw_error_code, address);
        else
        /* 处理用户态的缺页中断 */
                do_user_addr_fault(regs, hw_error_code, address);
}
NOKPROBE_SYMBOL(__do_page_fault);
```

3.3.3 用户态的缺页中断处理

`do_user_addr_fault()`传入的`hw_error_code`是页的错误码, 下面是其中的含义:

	set(=1)	clear(=0)
0位	保护性错误，越权访问产生异常	“存在位”为0，要访问的页面不在RAM中导致异常
1位	因为写访问导致异常(write)	因为读或者运行产生异常(read or execute)
2位	用户态(User Mode)	内核态(Kernel Mode)
3位	使用检测到的保留位	
4位	错误是指令获取	
5位	保护键阻止访问	

`do_user_addr_fault`具体实现:

```
/* Handle faults in the user portion of the address space */
static inline
void do_user_addr_fault(struct pt_regs *regs,
                        unsigned long hw_error_code,
                        unsigned long address)
{
    tsk = current;
    mm = tsk->mm;    /* 获取该进程的内存描述符mm */
    /* ... */
    /*
         * hw_error_code is literally the "page fault error code" passed to
         * the kernel directly from the hardware.  But, we will shortly be
         * modifying it in software, so give it a new name.
```

```
    */
    sw_error_code = hw_error_code;

    /* ... */

    vma = find_vma(mm, address); /* 通过address在内存描述符mm中查找vma */
    if (unlikely(!vma)) {    /* 假如不存在 */
            bad_area(regs, sw_error_code, address);  /* 访问非法地址 */
            return;
    }
    if (likely(vma->vm_start <= address))        /* 访问合法地址，跳转至good_area */
            goto good_area;
    if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
            bad_area(regs, sw_error_code, address);
            return;
    }
    if (sw_error_code & X86_PF_USER) {
            /*
             * Accessing the stack below %sp is always a bug.
             * The large cushion allows instructions like enter
             * and pusha to work. ("enter $65535, $31" pushes
             * 32 pointers and then decrements %sp by 65535.)
             */
        /* 访问了越界的栈空间 */
            if (unlikely(address + 65536 + 32 * sizeof(unsigned long) < regs->sp)) {
                    bad_area(regs, sw_error_code, address);
                    return;
            }
    }
    if (unlikely(expand_stack(vma, address))) {
            bad_area(regs, sw_error_code, address);
            return;
    }
```

首先尝试通过该进程的内存描述符`mm`获取`vma`即虚拟内存区域，假如不存在，则说明访问了非法的虚拟地址，返回`bad_area()`，同样假如是越界错误或者段权限错误也返回`bad_area()`.

假如访问的地址是合法，会跳转至`good_area`:

```
good_area:
        if (unlikely(access_error(sw_error_code, vma))) { // 根据页的错误类型与vma的访问权限是否匹配
                bad_area_access_error(regs, sw_error_code, address, vma);
                return;
        }

        /*
         * If for any reason at all we couldn't handle the fault,
         * make sure we exit gracefully rather than endlessly redo
```

```
         * the fault.  Since we never set FAULT_FLAG_RETRY_NOWAIT, if
         * we get VM_FAULT_RETRY back, the mmap_sem has been unlocked.
         *
         * Note that handle_userfault() may also release and reacquire
mmap_sem
         * (and not return with VM_FAULT_RETRY), when returning to userland
to
         * repeat the page fault later with a VM_FAULT_NOPAGE retval
         * (potentially after handling any pending signal during the return
to
         * userland). The return to userland is identified whenever
         * FAULT_FLAG_USER|FAULT_FLAG_KILLABLE are both set in flags.
         */
        fault = handle_mm_fault(vma, address, flags); /* 处理缺页的具体实现 */
        major |= fault & VM_FAULT_MAJOR;

        /* ... */

        check_v8086_mode(regs, address, tsk);
}
NOKPROBE_SYMBOL(do_user_addr_fault);
```

如果这个虚拟区的访问权限与引起错误的访问类型相匹配，假如是Huge Page(大页)的缺页中断，则调用`handle_mm_fault()`函数，而`handle_mm_fault()`调用`__handle_mm_fault()`完成具体操作:

3.3.4 __handle_mm_fault()

宏定义handle_mm_fault()，即__handle_mm_fault()函数，先生成一个指向页表项的指针，该页表项对应的虚拟地址范围包含了导致缺页的虚拟地址，然后以生成的指针作为参数调用函数handle_pte_fault()继续处理缺页。

```
static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
                unsigned long address, unsigned int flags)
{
        struct vm_fault vmf = {
                .vma = vma,
                .address = address & PAGE_MASK,
                .flags = flags,
                .pgoff = linear_page_index(vma, address),
                .gfp_mask = __get_fault_gfp_mask(vma),
        };
        unsigned int dirty = flags & FAULT_FLAG_WRITE;
        struct mm_struct *mm = vma->vm_mm;
        pgd_t *pgd;
        p4d_t *p4d;
        vm_fault_t ret;

        pgd = pgd_offset(mm, address);       /* 返回指定的mm的全局目录项的指针 */
        p4d = p4d_alloc(mm, pgd, address);  /* 在X86的4级页面机制中，不做任何操作，
直接返回pgd */
        if (!p4d)
```

```
                return VM_FAULT_OOM;

        vmf.pud = pud_alloc(mm, p4d, address);  /* 创建并分配一个Page Upper Directory指针 */
    /* ... */
    vmf.pmd = pmd_alloc(mm, vmf.pud, address);
    /*通过address和pud得出pmd，即中间层目录项。函数pmd_alloc()得到address所对应的中间层页目录项的地址。由于x86平台上没有使用中间页目录，所以实际上只是返回给定的pgd指针。*/
    if (!vmf.pmd)
                return VM_FAULT_OOM;
    /* ... */
    /* 根据vmf决定如何分配一个新的页面 */
        return handle_pte_fault(&vmf);
}
```

3.3.5 handle_pte_fault()分配页面

```
static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
{
        pte_t entry;

        if (unlikely(pmd_none(*vmf->pmd))) {
                /* 页中间目录不存在，即页表也为空 */
                vmf->pte = NULL;
        } else {
                /* ... */
        /* 页中间目录存在，通过address尝试获取页表(Page Table) */
                vmf->pte = pte_offset_map(vmf->pmd, vmf->address);
                vmf->orig_pte = *vmf->pte;
        /*
                 * some architectures can have larger ptes than wordsize,
                 * e.g.ppc44x-defconfig has CONFIG_PTE_64BIT=y and
                 * CONFIG_32BIT=y, so READ_ONCE cannot guarantee atomic
                 * accesses.  The code below just needs a consistent view
                 * for the ifs and we later double check anyway with the
                 * ptl lock held. So here a barrier will do.
                 */
                barrier();
                if (pte_none(vmf->orig_pte)) {
            /* 假如页中间目录存在，但页表不存在，vmf->pte置为NULL */
                        pte_unmap(vmf->pte);
                        vmf->pte = NULL;
                }
        }

    /* 假如vmf->pte为空，即尚未分配为缺失的页分配页表(Page Table) */
        if (!vmf->pte) {
                if (vma_is_anonymous(vmf->vma))
            /* 处理匿名文件映射的缺页 */
                        return do_anonymous_page(vmf);
                else
```

```
            /* 处理文件映射的缺页 */
                    return do_fault(vmf);
        }

    /* 页表已经建立，但不存在于物理内存之中 */
        if (!pte_present(vmf->orig_pte))
        /* 从磁盘交换区换入物理内存 */
                return do_swap_page(vmf);

        if (pte_protnone(vmf->orig_pte) && vma_is_accessible(vmf->vma))
                return do_numa_page(vmf);

        vmf->ptl = pte_lockptr(vmf->vma->vm_mm, vmf->pmd);
        spin_lock(vmf->ptl);
        entry = vmf->orig_pte;
    if (unlikely(!pte_same(*vmf->pte, entry)))
                goto unlock;
    /* 页表已经建立，且也贮存在物理内存中，因为写操作触发了缺页中断，即为COW的缺页中断 */
        if (vmf->flags & FAULT_FLAG_WRITE) {
                if (!pte_write(entry))
            /* 处理Copy On Write的Write部分的缺页中断 */
                    return do_wp_page(vmf);
                entry = pte_mkdirty(entry);
        }

    /* ... */
}
```

四、总结

本次实验的研究问题虽然只是do_page_fault函数，但是我发现如果仅仅研究do_page_fault函数本身，对于缺页处理的理解并不透彻。所以在研读了几篇关于do_page_fault分析的博文后我还是先去linux的指导课本上寻找内存管理相关的知识学习巩固了一番，之后再看do_page_fault的代码就透彻了许多。

其次如果仅仅研究do_page_fault函数，而不去理解Linux kernel按需调页的过程中调用的其他函数，我觉得也是不能完全理解do_page_fault函数的。因为do_page_fault函数在我做完本次研讨报告后的理解中，只是缺页中断服务的入口函数，必须理解其调用的一系列函数的意义才能理解do_page_fault函数。早期的内核版本的代码似乎不像4.6.20版本一样把do_page_fault的功能在这么多的函数调用中实现，比如《边干边学》里的do_page_fault函数就比较长，经粗略研究里面的功能应该和4.6.20里面调用的其他函数的功能一样，只是全部写在了一起，do_page_fault函数变得很长很长，为阅读增加了一定的难度。

之前的实验中对do_page_fault函数有了一个初步的认识，因为系统每发生一次缺页错误便会调用该函数，所以我们可以认为该函数的调用次数就是系统缺页的次数。这次研讨报告让我对实验三也有了更好的理解。

参考文献

1. 《边干边学——LINUX内核指导（第二版）》李善平 季江民 尹康凯 等编著 胡志刚 主审
2. 《缺页异常详解》（https://www.cnblogs.com/jikexianfeng/articles/5647994.html）

3. 《linux内存管理笔记(五）---缺页异常概述》（https://blog.csdn.net/u012489236/article/details/111415668?ops_request_misc=%25257B%252522request%25255Fid%252522%25253A%252522160907584616780310175083%252522%25252C%252522scm%252522%25253A%25252220140713.130102334.pc%25255Fblog.%252522%25257D&request_id=160907584616780310175083&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~first_rank_v1~rank_blog_v1-4-111415668.pc_v1_rank_blog_v1&utm_term=%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86%E4%BA%94）
4. 《Linux 内核源码分析-内存请页机制》（https://leviathan.vip/2019/03/03/Linux%E5%86%85%E6%A0%B8%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-%E5%86%85%E5%AD%98%E8%AF%B7%E9%A1%B5%E6%9C%BA%E5%88%B6/）