OSlab3--by：周延霖

吐槽&坑

1. 其实练习1比起后面的实现相对简单，主要就是一步步确认这个异常的确是缺页异常。do_pgfault主要是设置了层层if检查；
2. 练习1所涉及的get_pte函数以及pgdir_alloc_page函数在Lab2中都有所接触，所以用起来还算顺利，只是我写代码的时候并不会if(函数==NULL){cprintf"……"；goto failed;}，即我一般不会进行分配失败的检查，后来查看其他代码修改的时候才发现不写NULL检查的确不安全。所以以后还是要养成好的代码习惯。
3. do_pgfault这个函数实现的是扇面的换入，即swap_in，没有swap_out功能的实现；在考察swap_out功能实现的时候，一开始没想到alloc_pages这个lab2已经实现过的函数会被修改，并完成了这么重要的任务，所以没找到swap_out。后来在分析代码逻辑的时候，也就是从原理上看什么时候换出的时候才又转去看页面page的代码，发现这个alloc_pages已经和lab2不一样了，执行了swap_out(check_mm_struct,n,0);

实验部分

做完实验二后，大家可以了解并掌握物理内存管理中的连续空间分配算法的具体实现以及如 何建立二级页表。本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异 常处理机制，完成Page Fault异常处理和FIFO页替换算法的实现。实验原理最大的区别是在 设计了如何在磁盘上缓存内存页，从而能够支持虚存管理，提供一个比实际物理内存空间“更 大”的虚拟内存空间给系统使用。

实验目的如下

- 了解虚拟内存的Page Fault异常处理实现
- 了解页替换算法在操作系统中的实现

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成 Page Fault异常处理和FIFO页替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚 存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操 作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统 系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交 叉访问。如果大家有余力，可以尝试完成扩展练习，实现extended clock页替换算法。

练习0：前期准备

Lab1时修改了trap.c，init.c和kdegub.c；Lab2时修改了default_pmm.c和pmm.c 所以在Lab3开始之前要先对这些代码进行补充和修改。

练习1：给未被映射的地址映射上物理页（需要编程）

> 完成do_pgfault（mm/vmm.c）函数，给未被映射的地址映射上物理页。设置访问权限 的时候 需要参考页面所在 VMA 的权限，同时需要注意映射物理页时需要操作内存控制 结构所指定 的页表，而不是内核的页表。
>
> 请在实验报告中简要说明你的设计实现过程。请回答如下问题：

> - 请描述页目录项（Pag Director Entry）和页表（Page Table Entry）中组成部分对ucore 实现页替换算法的潜在用处。
> - 如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

代码实现

lab3在lab2的基础上，主要新增了以下功能：

1. kern_init总控函数中新增了vmm_init、ide_init、swap_init函数入口，分别完成了虚拟内存管理器、ide硬盘交互以及虚拟内存磁盘置换器的初始化。
2. 在trap.c的中断处理分发函数trap_dispatch中新增了对14号中断(页访问异常)的处理逻辑do_pgfault。do_pgfault除了接受前面提到的32位中断错误号和引起错误的线性地址，还接受了当前进程的mm_struct结构，用以访问当前进程的页表。do_pgfault对错误码进行了处理，判断其究竟是否是因为缺页造成的页访问异常；还是因为非法的虚拟地址访问、特权级的越级内存访问等错误引发的页异常，如果是后者就应该报错或者让进程直接奔溃掉。 如果发现访问的是一个合法的虚拟地址，则会进一步找到引起异常的线性地址所对应的二级页表项，判断其是真的不存在(pte中的每一位都是0)还是之前被暂时交换到了磁盘上(仅仅是P位为0)。 如果是真的不存在，则需要立即为其分配一个初始化后全新的物理页，并建立映射虚实关系。 如果是被暂时交换到了磁盘中，则需要将交换扇区中的数据重新读出并覆盖所分配到的物理页。 页异常中断属于异常中断的一种，当中断服务例程返回后，会重新执行引起页异常的那条指令，如果do_pafault实现正确，那么此时将能够正确的访问到虚拟地址对应的物理页，程序能正常的往下继续执行。 下面根据具体代码，来分析练习1的do_pgdault功能。

```
//这是第一步检查——检查是否是合法的虚拟地址
    // 试图从mm关联的vma链表块中查询，是否存在当前addr线性地址匹配的vma块
    struct vma_struct *vma = find_vma(mm, addr);
    // 全局页异常处理数自增1
    pgfault_num++;
    //判断范围
    if (vma == NULL || vma->vm_start > addr) {
        // 如果没有匹配到vma
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
```

```
//而后对trap_dispatch捕获到的异常错误码情况进行分析
// 页访问异常错误码有32位。位0为1 表示对应物理页不存在；位1为1 表示写异常(比如写了只读页)；位2为1 表示访问权限异常（比如用户态程序访问内核空间的数据）
    // 对3求模，主要判断bit0、bit1的值
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
        // bit0，bit1都为1，访问的映射页表项存在，且发生的是写异常
        // 说明发生了缺页异常
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        // bit0为0，bit1为1，访问的映射页表项不存在、且发生的是写异常
        if (!(vma->vm_flags & VM_WRITE)) {
```

```
            // 对应的vma块映射的虚拟内存空间是不可写的,权限校验失败
            cprintf("do_pgfault failed: error code flag = write AND not
present, but the addr's vma cannot write\n");
            // 跳转failed直接返回
            goto failed;
        }
        // 校验通过，则说明发生了缺页异常
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present */
        // bit0为1，bit1为0，访问的映射页表项存在，且发生的是读异常(可能是访问权限异
常)
        cprintf("do_pgfault failed: error code flag = read AND
present\n");
        // 跳转failed直接返回
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        // bit0为0，bit1为0，访问的映射页表项不存在，且发生的是读异常
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            // 对应的vma映射的虚拟内存空间是不可读且不可执行的
            cprintf("do_pgfault failed: error code flag = read AND not
present, but the addr's vma cannot read or exec\n");
            goto failed;
        }
        // 校验通过，则说明发生了缺页异常
    }
```

对这段异常错误码判断，可以总结如下：如果这个中断是要写一个存在的地址，或者要写一个不存在的可写地址，或者要读一个不存在的可读地址，都会启动缺页中断。

```
// 构造需要设置的缺页页表项的perm权限
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    // 构造需要设置的缺页页表项的线性地址(按照PGSIZE向下取整，进行页面对齐)
    addr = ROUNDDOWN(addr, PGSIZE);

// 用于映射的页表项指针（page table entry, pte)
    pte_t *ptep=NULL;
```

```
//以下是本次编码需要完成的地方:
//(1) try to find a pte, if pte's PT(Page Table) isn't existed, then
create a PT.

    // 获取addr线性地址在mm所关联页表中的页表项
    // 第三个参数=1 表示如果对应页表项不存在，则需要新创建这个页表项（这个是Lab2里面设
置的)
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("do_pgfault: get_pte failed\n");
```

```
            goto failed;
        }

    //(2) if the phy addr isn't exist, then alloc a page & map the phy addr
    with logical addr
     // 如果对应页表项的内容每一位都全为0，说明之前并不存在，需要设置对应的数据，进行线性地址
    与物理地址的映射
     if (*ptep == 0) {
      // 令pgdir指向的页表中，la线性地址对应的二级页表项与一个新分配的物理页Page进行虚实地
    址的映射
            if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
                cprintf("do_pgfault: pgdir_alloc_page failed\n");
                goto failed;
            }
        }


    //剩下部分为练习2的内容，为保证完整性，一并贴出。
        else {
            // 如果不是全为0，说明可能是之前被交换到了swap磁盘中
            if(swap_init_ok) {
                // 如果开启了swap磁盘虚拟内存交换机制
                struct Page *page=NULL;
                // 将addr线性地址对应的物理页数据从磁盘交换到物理内存中(令Page指针指向交
    换成功后的物理页)
                if ((ret = swap_in(mm, addr, &page)) != 0) {
                    // swap_in返回值不为0，表示换入失败
                    cprintf("do_pgfault: swap_in failed\n");
                    goto failed;
                }
                // 将交换进来的page页与mm->padir页表中对应addr的二级页表项建立映射关系
    (perm标识这个二级页表的各个权限位)
                page_insert(mm->pgdir, page, addr, perm);
                // 当前page是为可交换的，将其加入全局虚拟内存交换管理器的管理
                swap_map_swappable(mm, addr, page, 1);
                page->pra_vaddr = addr;
            }
            else {
                // 如果没有开启swap磁盘虚拟内存交换机制，但是却执行至此，则出现了问题
                cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
                goto failed;
            }
       }
```

问题回答

1. 页目录表和mm_struct结构对应，用于根据传入的线性地址索引对应的页表；页表项，即一个PTE用来描述一般意义上的物理页时，应该有PTE_P标记，即表示物理页存在；但当它用来描述一个被置换出去的物理页时，它被用来维护该物理页与swap磁盘上扇区的映射关系，此时没有PTE_P标记。页替换涉及到换入换出，换入时需要将某个虚拟地址对应于磁盘的一页内容读入到内存中，换出时需要将某个虚拟页的内容写到磁盘中的某个位置，因此页表项可以记录该虚拟页在磁盘中的位置，也为换入换出提供磁盘位置信息。

2. CPU会把产生异常的线性地址存储在CR2寄存器中，并且把表示页访问异常类型的error Code保存在中断栈中。

练习2：补充完成基于FIFO的页面替换算法（需要编程）

> 完成vmm.c中的do_pgfault函数，并且在实现FIFO算法的swap_fifo.c中完成map_swappable 和swap_out_vistim函数。通过对swap的测试。
>
> 请在实验报告中简要说明你的设计实现过程。 请在实验报告中回答如下问题：
>
> - 如果要在ucore上实现"extended clock页替换算法"请给你的设计方案，现有的 swap_manager 框架是否足以支持在ucore中实现此算法？如果是，请给你的设计方案。 如果不是，请给出你的新的扩展和基此扩展的设计方案
>   - 需要被换出的页的特征是什么？
>   - 在ucore中如何判断具有这样特征的页？
>   - 何时进行换入和换出操作？

关于代码

页面替换的过程可以分为页面换入和页面换出两个部分。页面换入部分在vmm.c中，承接在上一部分练习中完成的代码，而页面换出部分在swap_fifo.c中。

先来看页面换入的实现。

首先我们要先新分配一个物理页空间，保存通过 swap_in函数换出的磁盘中的数据，如果swap_in失败则报错返回。之后我们需要用page_insert这个函数将新申请的物理页面映射到线性地址上。最后因为我们是FIFO算法，把最近到达的页面排在队尾。而把页面排在队尾的功能在_fifo_map_swappable中。

因此可以编写代码如下

```
/* vmm.c */
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    switch (error_code & 3) {
    default:
    case 2:
        if (!(vma->vm_flags & VM_WRITE)) {
            cprintf("do_pgfault failed: error code flag = write AND not
present, but the addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1:
```

```
            cprintf("do_pgfault failed: error code flag = read AND
present\n");
            goto failed;
        case 0:
            if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
                cprintf("do_pgfault failed: error code flag = read AND not
present, but the addr's vma cannot read or exec\n");
                goto failed;
            }
        }
        uint32_t perm = PTE_U;
        if (vma->vm_flags & VM_WRITE) {
            perm |= PTE_W;
        }
        addr = ROUNDDOWN(addr, PGSIZE);
        ret = -E_NO_MEM;
        pte_t *ptep=NULL;
        ptep = get_pte(mm->pgdir,addr,1);
        if (*ptep == 0)
        {
            if(pgdir_alloc_page(mm->pgdir,addr,perm) == NULL)
            {
                cprintf("Error:pgdir_alloc_page\n");
                goto failed;
            }

        }
        else
        {
            if(swap_init_ok)
            {
                struct Page *page=NULL;
                if((ret = swap_in(mm,addr,&page))!=0)
                {
                     cprintf("Error:swap_in\n");
                    goto failed;
                }
                else
                {
                    page_insert(mm->pgdir,page,addr,perm);
                    swap_map_swappable(mm,addr,page,1);
                    page->pra_vaddr = addr;
                }
            }
            else
            {
                cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
                goto failed;
            }
       }
       ret = 0;
failed:
        return ret;
}
```

```
/* swap_fifo.c */
static int
_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);
    assert(entry != NULL && head != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 2: YOUR CODE*/
    //(1)link the most recent arrival page at the back of the
pra_list_head qeueue.
    list_add(head, entry);
    return 0;
}
```

既然有了IN，那肯定需要有OUT，OUT的功能主要在_fifo_swap_out_victim中实现。首先我们要找到需要被换出的页，并用一个指针le指示这个需要被换出的页。再用le2page找到对应的page，最后删除这个le指向的页并用之前找到的page替换参数中的ptr_page

代码实现如下

```
static int
_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int
in_tick)
{
     list_entry_t *head=(list_entry_t*) mm->sm_priv;
         assert(head != NULL);
     assert(in_tick==0);
     /* Select the victim */
     /*LAB3 EXERCISE 2: YOUR CODE*/
     //(1)  unlink the  earliest arrival page in front of pra_list_head
qeueue
     //(2)  assign the value of *ptr_page to the addr of this page
     /* Select the tail */
     list_entry_t *le = head->prev;
     assert(head!=le);
     struct Page *p = le2page(le, pra_page_link);
     list_del(le);
     assert(p !=NULL);
     *ptr_page = p;
     return 0;
}
```

关于问题

1. 现有的swap_manager框架支持在ucore中实现此算法

2. 具有这样特征的页应该满足如下特征

- 优先选择 Dirty Bit 为0，Access Bit 为0 的页
- 其次选择 Dirty Bit 为0，Access Bit 为1 的页
- 最后选择 Dirty Bit 为1，Access Bit 为0 的页

转换为代码如下

```
!(*ptep & PTE_A)&&!(*ptep & PTE_D); //最优
(*ptep & PTE_A) &&!(*ptep & PTE_D); //次优
!(*ptep & PTE_A)&& (*ptep & PTE_D); //最次
```

3. 缺页的时候换入，满页的时候换出

Challenge1：实现识别dirty bit的 extended clock页替 换算法（需要编程）

添加swap_extd_clk.c，其中包含如下结构体和函数：

```
struct swap_manager swap_manager_extended_clock =
{
 .name            = "extended clock swap manager",
 .init            = &_extd_clk_init,
 .init_mm         = &_extd_clk_init_mm,
 .tick_event      = &_extd_clk_tick_event,
 .map_swappable   = &_extd_clk_map_swappable,
 .set_unswappable = &_extd_clk_set_unswappable,
 .swap_out_victim = &_extd_clk_swap_out_victim,
 .check_swap      = &_extd_clk_check_swap,
};
```

仿照swap_fifo.c，逐个实现该结构体中的函数：

```
static int
_extd_clk_init(void)
{
    return 0;
}

list_entry_t pra_list_head;

static int
_extd_clk_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in _extd_clk_init_mm\n",mm->sm_priv);
     return 0;
}

static int
```

```
_extd_clk_tick_event(struct mm_struct *mm)
{ return 0; }

static int
_extd_clk_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
{
    //换入页的在链表中的位置并不影响，因此将其插入到链表最末端。
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && head != NULL);// 将新页插入到链表最后
    list_add(head -> prev, entry);//新页dirty bit标记为0.
    struct Page *ptr = le2page(entry, pra_page_link);
    pte_t *pte = get_pte(mm -> pgdir, ptr -> pra_vaddr, 0);
    *pte &= ~PTE_D;
    return 0;
}

static int
_extd_clk_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}
```

```
static int _extd_clk_swap_out_victim(struct mm_struct*mm,struct Page
**ptr_page,int in_tick)
{
//检查存储同一页表中虚拟内存块首页指针的循环链表；从表头开始
list_entry_t*head=(list_entry_t*)mm->sm_priv;
assert(head!=NULL);
assert(in_tick==0);

list_entry_t *p=list_next(head);
assert(p!=head);
//遍历链表，将最近未被访问且未被修改的页换出
while(1)
{
    //获取当前页的page结构
    struct Page *ptr=le2page(p,pra_page_link);
    //获取当前页对应二级页表项地址（用指针访问页表项）；二级页表项存储的是各个页的物理内
存地址
    pte_t *pte=get_pte(mm->pgdir,p->pra_vaddr,0);
    //如果该页dirty bit为0，可换出该页
    if(!(*pte&PTE_D))
    {
        //将页面从链表中删除
        list_del(p);
        assert(p!=NULL);
        /*将该页对应的page结构放到形参ptr_page指向的某处内存地址（在本函数栈帧以外，
因而函数返回后这个值理论上可完好保留）*/
```

```
            *ptr_page=ptr;
            return 0;
        }
        //如果dirty bit为1 ，将该页此位改为0，继续扫描
        else{
            *pte &= ~PTE_D;
        }
        p = list_next(p);
    }
    }
```

swap.c中对有效虚地址范围进行了限制，为0-0x1000，且ucore中虚地址是直接映射到物理地址的：

```
10  // the valid vaddr for check is between 0~CHECK_VALID_VADDR-1
11  #define CHECK_VALID_VIR_PAGE_NUM 5
12  #define BEING_CHECK_VALID_VADDR 0X1000
13  #define CHECK_VALID_VADDR (CHECK_VALID_VIR_PAGE_NUM+1)*0x1000
```

```
static int
_extd_clk_check_swap(void) {
    cprintf("write Virt Page c in extd_clk_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==4);
    cprintf("write Virt Page a in extd_clk_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==4);
    cprintf("write Virt Page d in extd_clk_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==4);
    cprintf("write Virt Page b in extd_clk_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==4);
    cprintf("write Virt Page e in extd_clk_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    cprintf("write Virt Page b in extd_clk_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    cprintf("write Virt Page a in extd_clk_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==6);
    cprintf("write Virt Page b in extd_clk_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==7);
    cprintf("write Virt Page c in extd_clk_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==8);
    cprintf("write Virt Page d in extd_clk_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==9);
```

```
    cprintf("write Virt Page e in extd_clk_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==10);
    cprintf("write Virt Page a in extd_clk_check_swap\n");
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==11);
    return 0;
}
```