LAB 3:缺页异常和页面置换

练习1：理解基于FIFO的页面替换算法

0. swap_init():设置好max_swap_offset，选择替换策略后跳转到对应替换策略初始化，设置swap_init_ok=1
1. trap(),发生异常，跳转至exception_handler(tf)
2. exception_handler(tf)，根据错误原因跳转至CAUSE_LOAD_PAGE_FAULT，或者CAUSE_STORE_PAGE_FAULT，后跳转至pgfault_handler(tf)
3. pgfault_handler(struct trapframe *tf)：跳转至 do_pgfault(check_mm_struct, tf->cause, tf->badvaddr)，传递错误原因，与错误的地址
4. 进入do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr)函数中

```
struct vma_struct *vma = find_vma(mm, addr);//找到addr对应的vma
//如果没找到，就退出
  uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }//设置perm
    addr = ROUNDDOWN(addr, PGSIZE);//将addr转化为页地址
    ptep = get_pte(mm->pgdir, addr, 1); //拿到它对应的的页表项
//如果成功拿到了，且swap_init_ok已经准备好
 struct Page *page = NULL;
 swap_in(mm, addr, &page) ;//从mm中的磁盘（定义的ide）中取出addr对应的页面，存到page中
page_insert(mm->pgdir,page,addr,perm);//构建相应的映射，刷新tlb表
swap_map_swappable(mm,addr,page,1);//将其设置为可交换，将页面的visited标志置为1
```

5. swap_in(mm, addr, &page) ,当交换区已满时

```
struct Page *result = alloc_page();//这里alloc_page()内部可能调用swap_out()
```

6. alloc_pages(size_t n)

```
page = pmm_manager->alloc_pages(n);//分配n个页
//如果有足够的物理页面，就不必换出其他页面
        //如果n>1，说明希望分配多个连续的页面，但是我们换出页面的时候并不能换出连续的页面
        //swap_init_ok标志是否成功初始化了
        if (page != NULL || n > 1 || swap_init_ok == 0) break;
 swap_out(check_mm_struct, n, 0);
        //调用页面置换的”换出页面“接口。这里必有n=1
```

7.swap_out(struct mm_struct *mm, int n, int in_tick)

```
 int r = sm->swap_out_victim(mm, &page, in_tick);//调用页面置换算法找到被换出的页面,存到page中
 swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page)；//尝试把要换出的物理页面写到硬盘上的交换区
 //成功换出则释放page,并刷新TLB
 *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
                    free_page(page);
  tlb_invalidate(mm->pgdir, v);       。
```

返回swap_in(mm, addr, &page) ;继续执行新页面的进入

练习2：深入理解不同分页模式的工作原理

此时我们使用的是三级页表，

```
// Sv39 virtual address:
// +----9----+----9---+----9---+---12--+
// |  VPN[2] | VPN[1] | VPN[0] | PGOFF |
// +---------+----+---+--------+-------+
```

因此在获得后两级页内偏移时，会有相似的处理过程

```
pde_t *pdep1 = &pgdir[PDX1(la)];//the index of page directory entry of VIRTUAL ADDRESS la
    //此时pgdir可看作基地址
    //页目录索引（PDX）
    //PDX1获取 VPN[2]的偏移量，从pgdir中偏移，即可获取下一级的基地址VPN[1]
    if (!(*pdep1 & PTE_V)) {
        //如果此时有效位无效，构建一个新的页
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        //如果此时不允许构建新页面，返回NULL
        set_page_ref(page, 1);//设置此时page被引用的次数为1
        uintptr_t pa = page2pa(page);//得到page的物理地址
        memset(KADDR(pa), 0, PGSIZE);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
        //此时创建一个User can access，且有效的页表项
    }
```

此处中，pgdir为页目录表的基址，通过PDX1(la)获得其页内偏移量，从而取出下一级页表的基址，如果此时没有取到该页目录项，则创建一个page并分配给它

```
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
     //PDX0获取 VPN[1]的偏移量，从pgdir中偏移，即可获取下一级的基地址VPN[0]
    //pde_t *pdep0 = &((pde_t *)(PDE_ADDR(*pdep1)))[PDX0(la)];
    if (!(*pdep0 & PTE_V)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
```

```
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);//获取page的物理地址
        memset(KADDR(pa), 0, PGSIZE);
//      memset(pa, 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
```

该步骤与逻辑与上述类似

```
 return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
```

从VPN[0]获取物理地址，并返回页表项

练习3：给未被映射的地址映射上物理页

代码补全如下

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);//找到addr对应的vma

    pgfault_num++;//记录page fault number

    //If the addr is in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                         //PT(Page Table) isn't existed, then
                                         //create a PT.
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {

        if (swap_init_ok) {
            struct Page *page = NULL;
            // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
            //(1）According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            swap_in(mm, addr, &page) ;//从mm中的磁盘中取出addr对应的页面，存到page中

            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
```

```
            page_insert(mm->pgdir,page,addr,perm);//将此时的page保存在内存中，并构建相应的映射

            //(3) make the page swappable.
            swap_map_swappable(mm,addr,page,1);//将其设置为可交换
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
   }

   ret = 0;
failed:
    return ret;
}
```

3.1请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处

页目录项（Page Directory Entry）和页表项（Page Table Entry）在ucore中主要用于转换虚拟地址到物理地址。此外，这两个数据结构中还包含了一些控制位，比如存在位、读/写位、用户/超级用户位等等，这些位可以用来控制对应的物理页的访问权限以及其存在状态。在实现页替换算法时，可以利用这些控制位来确定哪些页可以被替换。例如，如果一个页表项的存在位为0，那么对应的物理页就可以被替换。

3.2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?

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，那么硬件会再次触发缺页异常，然后跳转到异常处理程序来处理这个异常。如果异常处理程序不能处理这个异常，那么ucore可能会崩溃

3.3数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系?如果有，其对应关系是啥?

Page结构的数组元素与页表中的页目录项和页表项是有对应关系的。Page结构的数组索引是物理页的编号，而页目录项和页表项中的物理页基地址则对应着这个编号。也就是说，Page结构数组中的每一项都代表了一个物理页，而页目录项和页表项则决定了这个物理页如何被映射到虚拟地址空间中

练习4：补充完成Clock页替换算法

```
list_entry_t pra_list_head, *curr_ptr;
/*
 * (2) _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr of pra_list_head.
 *              Now, From the memory control struct mm_struct, we can access FIFO PRA
 */
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: YOUR CODE*/
     // 初始化pra_list_head为空链表
     list_init(&pra_list_head);
     // 初始化当前指针curr_ptr指向pra_list_head，表示当前页面替换位置为链表头
```

```
     curr_ptr=&pra_list_head;
     // 将mm的私有成员指针指向pra_list_head，用于后续的页面替换算法操作
     mm->sm_priv=&pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
/*
 * (3)_fifo_map_swappable: According FIFO PRA, we should link the most recent
arrival page at the back of pra_list_head qeueue
 */
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
    list_entry_t *entry=&(page->pra_page_link);
    curr_ptr=(list_entry_t*)mm->sm_priv;//sm_priv中保存了，这了链表的起点
    assert(entry != NULL && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // link the most recent arrival page at the back of the pra_list_head qeueue.
    // 将页面page插入到页面链表pra_list_head的末尾
    list_add_before(curr_ptr, entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited=1;
    return 0;
}
/*
 *  (4)_fifo_swap_out_victim: According FIFO PRA, we should unlink the  earliest
arrival page in front of pra_list_head qeueue,
 *                            then set the addr of addr of this page to ptr_page.
 */
static int
_clock_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 */
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  set the addr of addr of this page to ptr_page
     curr_ptr=head;
    while (1) {
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 编写代码
        curr_ptr = list_next(curr_ptr);

        if (curr_ptr == head)
            curr_ptr = list_next(curr_ptr);

        struct Page * page = le2page(curr_ptr, pra_page_link);
        if (page->visited == 1)
            page->visited = 0;
        else {
            cprintf("curr_ptr 0xffffffff%08x\n", curr_ptr);
            list_del(curr_ptr);
```

```
            cprintf("curr_ptr 0xffffffff%08x\n", curr_ptr);
            (*ptr_page)=page;
            break;
        }

        // 获取当前页面对应的Page结构指针

        // 如果当前页面已被访问，则将visited标志置为0，表示该页面已被重新访问
    }
    return 0;
}
```

4.1比较Clock页替换算法和FIFO算法的不同

- 此时链表中除了 pra_list_head,指向链表头节点 ，新增了curr_ptr指向链表当前节点。
- 在_clock_init_mm时，也需要初始化当前curr_ptr为链表头
- 在_clock_map_swappable中，插入一个节点时需要设置page->visited=1，将页面page插入到页面链表pra_list_head的末尾，list_add_before(curr_ptr, entry)
- 在_clock_swap_out_victim中，需要通过一个循环，清除各个节点的visited，最后找到一个合适的节点。

练习5：阅读代码和实现手册，理解页表映射方式相关知识

5.1如果我们采用”一个大页“ 的页表映射方式，相比分级页表，有什么好处、优势，有什么坏处、风险?

- 好处：减少了页表级数，降低了访问内存时的延迟。因为在分级页表中，每一级的页表访问都可能会导致一个内存访问，而内存访问的速度远低于寄存器访问。使用大页可以减少页表的级数，从而提高内存访问的效率。
- 缺点：如果一个大页中的只有一小部分被使用，那么剩余的部分就会浪费。在分级页表中，可以通过分配更小的页来避免这种浪费