《操作系统原理》lab4实验报告

专业：管理学院会计学（辅修）

姓名：莫书琪

学号：17318086

1.实验目的

- 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

2.实验过程

练习1：给未被映射的地址映射上物理页

do_pgfault中填写的部分

`do_pgfault`函数的作用是处理缺页异常。实验中已实现的代码体现了：函数根据从CPU的控制寄存器CR2中获取页访问异常的物理地址，首先判断当前访问的虚拟页是否在已经分配的虚拟页中，然后根据errorCode的错误类型判断是否满足正确的读写权限，如果在此范围内并且权限也正确，则认为这是一次合法访问。

练习1填写的代码作用是当程序判断为合法访问后，进一步建立虚实对应关系。程序查找当前虚拟地址所对应的页表项，如果这个页表项所对应的物理页不存在，则分配一个空闲的内存页，并修改页表完成虚地址到物理地址的映射。

```
    // 1.查找当前虚拟地址所对应的页表项
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }
    // 2.如果这个页表项所对应的物理页不存在，则分配一个空闲的内存页
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {  // 权限不够
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
```

代码中涉及的关键数据结构：负责管理虚拟内存页面的结构体`mm_struct`，负责存储虚拟页面信息的结构体`vma_struct`和负责存储物理页面信息的结构体`Page`。

```
// 管理虚拟内存页面的结构体
struct mm_struct {
    list_entry_t mmap_list;         // 双向链表头
    struct vma_struct *mmap_cache; // 指向当前正在使用的虚拟内存空间
    pde_t *pgdir;                   // 指向所维护的一级页表
    int map_count;                  // 记录mmap_list里面链接的vma_struct的个数
    void *sm_priv;                  // 指向用来链接记录页访问情况的链表头
};
```

```
// 存储虚拟页面信息的结构体
struct vma_struct {
    struct mm_struct *vm_mm;       // 管理该虚拟页的mm_struct
    uintptr_t vm_start;            // 虚拟页起始地址，包括当前地址
    uintptr_t vm_end;              // 虚拟页终止地址，不包括当前地址（地址前闭后开）
    uint32_t vm_flags;             // 相关标志位
    list_entry_t list_link;        // 用于连接各个虚拟页的双向指针
};

// 存储物理页面信息的结构体，与实验3相比增加了最后两个变量
struct Page {
    int ref;
    uint32_t flags;
    unsigned int property;
    list_entry_t page_link;
    list_entry_t pra_page_link;      // 用于连接上一个和下一个已分配且可交换的物理页
    uintptr_t pra_vaddr;             // 用于保存该物理页所对应的线性地址
};
```

问题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。

页替换时需要把磁盘的内容写到内存中或把内存的内容写到磁盘中。页目录项的作用是索引页表，页表项的作用是记录虚拟页在磁盘中的位置，为页替换提供位置信息和操作权限信息。

问题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?

出现页访问异常时，硬件需要对中断进行响应。

1. 保护现场：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
2. 关中断，保护断点
3. 根据入口地址进入中断服务处理中的缺页服务例程：trap - trap_dispatch - pgfault_handler - do_pgfault
4. 恢复现场，中断返回

练习2：补充完成基于FIFO的页面替换算法

do_pgfault中填写的部分

ucore的缺页异常会交给`do_pgfault`函数处理，当该函数判断物理页存在但不在内存中时，这说明可能是之前被交换到了磁盘中，此时需要将其换入内存以进行正常的数据访问。`do_pgfault`函数中进行页面替换的思路是：程序首先判断是否对交换机制进行了正确的初始化，然后将虚拟页对应物理页从外存中换入内存，并建立相应映射关系，最后将换入的物理页设置为允许被换出和保存其线性地址。

```
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    // 如果物理页存在但不在内存中，说明可能是之前被交换到了磁盘中
    else {
        // 1.如果交换机制正确初始化了
        if(swap_init_ok) {
```

```
            struct Page *page=NULL;
            // 2.将线性地址addr对应的页面mm换入到新的物理页page中
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            // 3.建立映射关系：关联虚拟地址（mm->pgdir页表中对应addr的二级页表项）与物理页
page
            page_insert(mm->pgdir, page, addr, perm);
            // 4.设置当前页为可交换
            swap_map_swappable(mm, addr, page, 1);
            // 5.保存物理页page对应的线性地址addr
            page->pra_vaddr = addr;
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
```

把磁盘物理页换入到内存中的过程

`do_pgfault`函数代码涉及到了把磁盘物理页换入到内存中的关键过程：主要通过调用`swap_in`函数实现，`swap_in`函数会进一步调用`alloc_page`函数进行物理页分配，一旦没有足够的物理页，则会使用`swap_out`函数将当前物理空间的某一页换出到外存，换出页面的选择算法是在`_fifo_swap_out_victim`函数中实现的`FIFO`算法。

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     // 1.分配一个新的物理页result
     struct Page *result = alloc_page();
     assert(result!=NULL);
     pte_t *ptep = get_pte(mm->pgdir, addr, 0);
     int r;
     // 2.将磁盘中读入的一整个物理页数据，写入result
     if ((r = swapfs_read((*ptep), result)) != 0)
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n",
(*ptep)>>8, addr);
     // 3.令参数ptr_result指向已被换入内存中的result Page结构
     *ptr_result=result;
     return 0;
}

struct Page *
alloc_pages(size_t n) {
    struct Page *page=NULL;
    bool intr_flag;

    while (1)
    {
        // 分配内存时要关中断
        local_intr_save(intr_flag);
```

```
        {
            page = pmm_manager->alloc_pages(n);
        }
        local_intr_restore(intr_flag);
        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        extern struct mm_struct *check_mm_struct;
        // 尝试着将某一物理页置换到swap磁盘交换扇区中，以腾出一个新的物理页来
        swap_out(check_mm_struct, n, 0);
    }
    return page;
}

int
swap_out(struct mm_struct *mm, int n, int in_tick)
{
     int i;
     for (i = 0; i != n; ++ i)
     {
          uintptr_t v;
          struct Page *page;
          // 1.swap置换管理器挑选被置换到swap磁盘扇区的page
          int r = sm->swap_out_victim(mm, &page, in_tick);
          if (r != 0) {
              cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
              break;
          }

          v=page->pra_vaddr;                          // 获得挑选出来的物理页的虚拟地址
          pte_t *ptep = get_pte(mm->pgdir, v, 0);    // 获得二级页表项
          assert((*ptep & PTE_P) != 0);

          // 2.将其写入swap磁盘
          // page->pra_vaddr/PGSIZE = 虚拟地址对应的二级页表项索引(前20位)；
          // (page->pra_vaddr/PGSIZE) + 1 （+1为了在页表项中区别 0 和 swap 分区的映
射）
          // ((page->pra_vaddr/PGSIZE) + 1) << 8，为了构成swap_entry_t的高24位
          if (swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page) != 0) {
              cprintf("SWAP: failed to save\n");
              sm->map_swappable(mm, v, page, 0);
              continue;
          }
          else {    // 交换成功
              cprintf("swap_out: i %d, store page in vaddr 0x%x to disk swap
entry %d\n", i, v, page->pra_vaddr/PGSIZE+1);
              *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
              free_page(page);
          }
          // 3.由于对应二级页表项出现了变化，刷新TLB快表
          tlb_invalidate(mm->pgdir, v);
     }
     return i;
}
```

_fifo_map_swappable

该函数的作用是将当前的物理页面插入到 FIFO 算法中维护的可被交换出去的物理页面链表中的末尾，其中双向循环链表在末尾插入等于在头节点前面插入。

```
static int
_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
    // 1.获取mm关联的链表头
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    // 2.获取page用于组织成链表的节点
    list_entry_t *entry=&(page->pra_page_link);
    assert(entry != NULL && head != NULL);
    // 3.将当前指定的物理页插入到链表的末尾
    list_add(head, entry);
    return 0;
}
```

函数调用关系：do_pgfault 函数调用 swap_map_swappable 函数将指定物理页设置为可交换，该函数会进一步调用 swap_manager 结构体里面封装的 _fifo_map_swappable 函数。

```
int
swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
     return sm->map_swappable(mm, addr, page, swap_in);
}

struct swap_manager swap_manager_fifo =
{
     .name            = "fifo swap manager",
     .init            = &_fifo_init,
     .init_mm         = &_fifo_init_mm,
     .tick_event      = &_fifo_tick_event,
     .map_swappable   = &_fifo_map_swappable,
     .set_unswappable = &_fifo_set_unswappable,
     .swap_out_victim = &_fifo_swap_out_victim,
     .check_swap      = &_fifo_check_swap,
};
```

_fifo_swap_out_victim

该函数的作用是选择需要被换出的页面。在 FIFO 算法中，按照物理页面换入到内存中的顺序建立了一个链表，而且每次访问一个页都是插入到头节点的后面，所以链表头的前面就是最早进入的物理页面，即需要被换出的页面。因此该函数的实现思路是：将链表头的物理页面取出，然后删掉对应的链表项。

```
static int
_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int
in_tick)
{
     // 1.获取mm关联的链表头
     list_entry_t *head=(list_entry_t*) mm->sm_priv;
     assert(head != NULL);
     assert(in_tick==0);
```

```
     // 2.获取链表头前面的页（最早访问的页）
     list_entry_t *le = head->prev;
     assert(head!=le);
     // 3.获取该页的page结构
     struct Page *p = le2page(le, pra_page_link);
     // 4.删除即将被换出的物理页
     list_del(le);
     assert(p !=NULL);
     // 5.将这一页地址保存到ptr_page中
     *ptr_page = p;
     return 0;
}
```

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。并需要回答如下问题：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？

现有的swap_manager框架可以支持实现时钟页替换算法。设计方案如下代码所示：

```
static int
_extended_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);
    for (int i = 0; i < 3; i++) {
        list_entry_t *le = head->prev;
        assert(head!=le);
        while (le != head) {
            struct Page *p = le2page(le, pra_page_link);
            pte_t* ptep = get_pte(mm->pgdir, p->pra_vaddr, 0);
            if (!(*ptep & PTE_A) && !(*ptep & PTE_D)) {
                list_del(le);
                assert(p != NULL);
                *ptr_page = p;
                return 0;
            }
            if (!i) {
                *ptep &= ~PTE_A;
            }
            else if (i == 1) {
                *ptep &= ~PTE_D;
            }
            le = le->prev;
            tlb_invalidate(mm->pgdir, le);
        }
    }
    return -1;
}

struct swap_manager swap_manager_fifo =
{
     .name            = "extended_clock swap manager",
     .init            = &_fifo_init,
```

```
    .init_mm         = &_fifo_init_mm,
    .tick_event      = &_fifo_tick_event,
    .map_swappable   = &_fifo_map_swappable,
    .set_unswappable = &_fifo_set_unswappable,
    .swap_out_victim = &_extended_clock_swap_out_victim,
    .check_swap      = &_fifo_check_swap,
};
```

- *需要被换出的页的特征是什么?*
    1. 没被访问过也没被修改过的页
    2. 访问了但没修改的页或者是修改了但没被访问的页
    3. 访问了也修改了的页
- *在ucore中如何判断具有这样特征的页?*
    - 页表项Dirty Bit为0表示没有被修改和Access Bit为0表示没有被访问
    - `!(*ptep & PTE_A) && !(*ptep & PTE_D)`表示没被访问过，也没被修改过
    - `!(*ptep & PTE_A) && (*ptep & PTE_D)`表示没被访问过，但被修改过
    - `(*ptep & PTE_A) && !(*ptep & PTE_D)`表示被访问过，但没被修改过
    - `(*ptep & PTE_A) && (*ptep & PTE_D)`表示被访问过，也被修改过
- *何时进行换入和换出操作?*
    - 在出现缺页异常时换入，在物理页帧满时换出

运行结果截图

练习1

```
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31964
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
```

练习2

```
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 0, total is 7
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
```

3.实验感想

通过本次实验进一步理解了操作系统是如何实现内存管理的。