uCore实验三报告

黄郭斌 计科1502班 201507010206

一. 内容

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。

二. 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

三. 实验分析及结果

0. 练习零

> 本实验依赖实验1/2。请把你做的实验1/2的代码填入本实验中代码中有“LAB1”/“LAB2”的注释相应部分。并确保编译通过。

需要填充的文件是default_pmm.c、pmm.c、trap.c、;不需要在以前的基础上再额外修改,直接根据对比复制过来即可。

1. 练习一

- **具体的算法思想:**

  当启动分页机制以后，如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误（比如写一个只读页或用户态程序访问内核态的数据等），就会发生页错误异常。产生页面异常的原因主要有:

  - 目标页面不存在（页表项全为0，即该线性地址与物理地址尚未建立映射或者已经撤销）;
  - 相应的物理页面不在内存中（页表项非空，但Present标志位=0，比如在swap分区或磁盘文件上）;
  - 访问权限不符合（此时页表项P标志=1，比如企图写只读页面）;

  当出现上面情况之一,那么就会产生页面page fault(#PF)异常。产生异常的线性地址存储在CR2中,并且将是page fault的产生类型保存在 error code 中。do_pgfault()函数从CR2寄存器中获取页错误异常的虚拟地址，根据error code来查找这个虚拟地址是否在某一个VMA的地址范围内，如果在范围内那么就给它分配一个物理页。

  具体实现如下:

  虚拟内存空间(VMA)是描述应用程序对虚拟内存的结构体:

```
struct vma_struct {
        // the set of vma using the same PDT`
        struct mm_struct *vm_mm;
        uintptr_t vm_start;      // start addr of vma
        uintptr_t vm_end;      // end addr of vma
        uint32_t vm_flags;     // flags of vma
        //linear list link which sorted by start addr of vma
        list_entry_t list_link;
};
```

- `vm_start`和`vm_end`描述的是一个合理的地址空间范围（即严格确保 vm_start < vm_end的关系）；
- `list_link`是一个双向链表，按照从小到大的顺序把一系列用`vma_struct`表示的虚拟内存空间链接起来，并且还要求这些链起来的`vma_struct`应该是不相交的，即vma之间的地址空间无交集；
- `vm_flags`表示了这个虚拟内存空间的属性，目前的属性包括

  ```
  #define VM_READ 0x00000001   //只读
  #define VM_WRITE 0x00000002  //可读写
  #define VM_EXEC 0x00000004   //可执行
  ```

- `vm_mm`是一个指针，指向一个比`vma_struct`更高的抽象层次的数据结构`mm_struct`而这`mm_struct`包含所有虚拟内存空间的共同属性，如下：

  ```
  struct mm_struct {
      // linear list link which sorted by start addr of vma
      list_entry_t mmap_list;
      // current accessed vma, used for speed purpose
      struct vma_struct *mmap_cache;
      pde_t *pgdir; // the PDT of these vma
      int map_count; // the count of these vma
      void *sm_priv; // the private data for swap manager
  };
  ```

- `mmap_list`是双向链表头，链接了所有属于同一页目录表的虚拟内存空间
- `mmap_cache`是指向当前正在使用的虚拟内存空间
- `pgdir`所指向的就是`mm_struct`数据结构所维护的页表
- `map_count`记录`mmap_list`里面链接的`vma_struct`的个数
- `sm_priv`指向用来链接记录页访问情况的链表头

- **具体实现:**

```
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) { //目标页面不存在，失败
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }

    if (*ptep == 0) {   //权限不够，也是失败！
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
```

```
        }
    }
    else {   //页表项非空，可以尝试换入页面
        if(swap_init_ok) {
            struct Page *page=NULL;//根据mm结构和addr地址，尝试将硬盘中的内容换入至page中
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            page_insert(mm->pgdir, page, addr, perm);//建立虚拟地址和物理地址之间的对应关系
            swap_map_swappable(mm, addr, page, 1);//将此页面设置为可交换的
            page->pra_vaddr = addr;
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
```

- **问题1：** 页目录项（pgdir）作为一个双向链表存储了目前所有的页的物理地址和逻辑地址的对应关系，即在实内存中的所有页，替换算法中被换出的页从pgdir中选出。页表（pte）则存储了替换算法中被换入的页的信息，替换后会将其映射到一物理地址。
- **问题2:** 产生页访问异常后，CPU把引起页访问异常的线性地址装到寄存器CR2中，并给出了出错码errorCode，说明了页访问异常的类型。ucore OS会把这个值保存在struct trapframe 中tf_err成员变量中。而中断服务例程会调用页访问异常处理函数do_pgfault进行具体处理。

2. 练习二

页错误异常发生时，有可能是因为页面保存在swap区或者磁盘文件上造成的，所以我们需要通过页面分配解决这个问题。 页面替换主要分为两个方面，页面换出和页面换入。

- 页面换入主要在上述的`do_pgfault()`函数实现；
- 页面换出主要在`swap_out_vistim()`函数实现。

这里换入在练习1已经完成了，这里就主要介绍换出。 `FIFO`替换算法会维护一个队列，队列按照页面调用的次序排列，越早被加载到内存的页面会越早被换出。 具体实现的函数如下： 首先是`_fifo_map_swappable()`，它的主要作用是将最近被用到的页面添加到算法所维护的次序队列。

```
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);
        list_add(head, entry); //将最近用到的页面添加到次序队尾
        return 0;
    } 1234567
```

然后是`_fifo_swap_out_victim()`函数是用来查询哪个页面需要被换出，它的主要作用是用来查询哪个页面需要被换出。

```
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);
    list_entry_t *le = head->prev;  //用le指示需要被换出的页
    assert(head!=le);
    struct Page *p = le2page(le, pra_page_link);//le2page宏可以根据链表元素获得对应的Page指针p

    list_del(le);      //将进来最早的页面从队列中删除
    assert(p !=NULL);
    *ptr_page = p; //将这一页的地址存储在ptr_page中
    return 0;
}
```

- **问题1：**

  目前的swap_manager框架足以支持在ucore中实现extended clock算法。

  在mmu.h中，有以下定义：

  `#define PTE_A 0x020 //Accessed`

  修改_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)函数:

  1. 首先让le被换出的页指向队列首;
  2. 接着循环判断当前页的PTE_A是否被设置为访问;若没有被设置为访问,则从队列中换出;若被访问顺着链表往下搜索;

  具体实现:

  ```
  list_entry_t *le = head->next;
  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))
  { //未被访问
  list_del(le);
  assert(p != NULL);
  *ptr_page = p;
  return 0;
   }
  *ptep ^= PTE_A;
  le = le->next;
  }
  ```

  > ① PTE_A为0的页可以被换出。
  >
  > ② 不断判断当前页的PTE_A是否被访问,如果未被访问则换出。
  >
  > ③ 当需要调用的页不在页表中时，并且在页表已满的情况下，需要进行换入和换出操作。

  运行结果：

  make qemu之后的结果:

```
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
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
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [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
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
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
count is 7, total is 7
check_swap() succeeded!
```

四. 实验体会和思考题

通过这次关于虚拟内存管理的实验，我大致知道了缺页错误出现的时候我们需要如何处理。通过实现了一个FIFO替换算法，了解了页面置换算法在UCore中的实现。

最后，通过练习2，使我对clock替换算法有了初步的了解,这个算法主要是维护一个访问位,在ucore中使用PTE_A实现,该算法主要替换未被访问过的页。