Lab3 实验报告

虚拟内存是操作系统为程序提供的物理内存抽象。分页机制建立后，cpu所看到的地址就不再是物理地址，而是虚拟地址。虚拟地址通常与物理地址不相等，存在某种映射关系，当对虚拟内存地址进行访问时，虚拟地址将被自动转换为物理内存地址。

需要注意的是，虚拟地址并不一定有对应的物理地址，在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在访问某虚拟内存地址时，操作系统才分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为**按需分页**（demand paging）。将不经常访问的数据保存在硬盘上，当需要访问这些数据时再将其从硬盘读入内存中使用，这被称为**页换入换出**（page swap in/out）。通过这样的方式，就可以为当前正在运行的程序或经常访问的数据提供更大的内存空间。

Lab3需要完成的是完成Page Fault异常处理和FIFO页替换算法的实现。

练习一：给未被映射的地址映射上物理页

1.页访问异常

实现虚拟内存管理的一个关键是page fault异常处理，其过程中主要涉及到函数do_pgfault的具体实现。当程序执行过程中产生了无法实现虚拟地址到物理地址的映射时，就会产生页访问异常，执行相应的中断服务例程，在处理异常时，操作系统就会完成按需分页，页换入换出等内存管理工作。页访问异常主要有以下类型：

- 目标页帧不存在（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)；
- 相应的物理页帧不在内存中（页表项非空，但Present标志位=0)
- 不满足访问权限(此时页表项P标志=1，但低权限的程序试图访问高权限的地址空间，或者有程序试图写只读页面).

产生以上异常时，CPU会把产生异常的线性地址存储在CR2（页故障线性地址寄存器）中，并且把表示页访问异常类型的值（简称页访问异常错误码，errorCode）保存在中断栈中，调用中断服务例程进行处理。中断服务例程将调用页访问异常处理函数do_pgfault进行具体处理。按需分页，页的替换均在此函数中实现。发生页访问异常时的调用过程如下：

trap--> trap_dispatch-->pgfault_handler-->do_pgfault

2.mm_struct和vma_struct

为了正常进行页访问异常处理，ucore需要mm_struct和vma_struct数据结构来描述不在物理内存中的“合法”虚拟页，根据其对地址的描述进行具体的处理。其中mm_struct描述了所有虚拟内存空间的共同属性，vma_struct描述程序对虚拟内存的需求。

mm_struct

mm_struct链接了所有属于同一页目录表的虚拟内存空间，其定义如下：

```
struct mm_struct {
    list_entry_t mmap_list;             //链接虚拟内存空间
    struct vma_struct *mmap_cache;      //当前正在使用的虚拟内存空间
    pde_t *pgdir;                       //页目录
    int map_count;                      //所链接的虚拟内存空间的个数
    void *sm_priv;                      //用来链接记录页访问情况的链表
};
```

涉及mm_struct的操作函数只有mm_create和mm_destroy两个函数，用于创建该变量和删除变量并释放空间。

vma_struct

vma_struct描述了一个合法的虚拟地址空间，定义如下

```
struct vma_struct {
    // the set of vma using the same PDT
    struct mm_struct *vm_mm;
    uintptr_t vm_start;                 //起始位置
    uintptr_t vm_end;                   //结束位置
    uint32_t vm_flags;                  //标志
    list_entry_t list_link;             //双向链表，按照从小到大的顺序链接vma_struct表示的虚拟内存空间
};
```

其中vm_flags表示了虚拟内存空间的属性，属性包括：

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

涉及vma操作的函数有三个，分别为：

- vma_create
- insert_vma_struct
- find_vma

vma_create函数根据输入参数vm_start，vm_end，vm_flags来创建并初始化描述一个虚拟内存空间的vma_struct结构变量。insert_vma_struct函数完成把一个vma变量插入到所属的mm变量中的mmap_list双向链表中。find_vma根据输入参数addr和mm变量，查找在mm变量中的mmap_list双向链表中的vma，找到的vma所描述的虚拟地址空间包含需要查找的addr。

3.给未被映射的地址映射上物理页

访问一个未被映射的地址所产生的异常是页访问异常的一种，在页访问异常处理时遇到这种情况，将会为该地址建立到物理页的映射。这个工作就在处理页访问异常的do_pgfault函数中完成。该函数定义如下：

```
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr);
```

mm是使用相同页目录的vma的管理结构，error_code是页访问异常的错误码，帮助确定页访问异常的原因，addr是导致页访问异常的地址。其中错误码包含以下信息：

- P flag （bit 0）：表明异常是否是由于访问不存在的物理页而产生(0)
- W/R flag (bit 1)： 表明导致异常的内存访问是由于读 (0) 还是写 (1).
- U/S flag (bit 2) ： 表明发生异常时是处于用户态 (1) 还是更高的特权态 (0)

在do_pgfault进行页访问异常处理中，首先要做的是调用find_vma寻找包含了导致访问异常的地址的虚拟内存空间vma，如果这个虚拟地址在分配好的虚拟内存空间中不存在，则这是一次非法访问，如果存在，则做进一步的处理。

```
    int ret = -E_INVAL;                 //error.h中定义的非法参数错误码
    //寻找包含了addr的vma
    struct vma_struct *vma = find_vma(mm, addr);
    pgfault_num++;
    //没有找到则为非法访问
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
```

如果找到了访问异常的虚拟地址所在的vma，将根据错误码进行对应的处理，这里根据错误码进行时处理不涉及特权级的判断：

```
    //根据错误码给出错误信息
    switch (error_code & 3) {
    default:
    //错误码为3，W/R=1，P=1：物理页存在，写操作，可能是权限错误
    //进入case2
    case 2:
    //错误码为2，W/R=1，P=0：物理页不存在，写操作，判断虚拟地址空间是否有写权限
        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:
    //错误码为1，W/R=0，P=1：物理页存在，读操作，可能是权限错误
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0:
    //错误码为0，W/R=0，P=0：物理页不存在，读操作，判断虚拟地址空间是否有读或执行权限
        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;
        }
    }
```

经过以上根据错误码的处理，由于权限问题导致的错误将直接返回。如果没有返回则说明地址访问满足权限要求，但访问的虚拟地址对应的物理页不存在，这种情况下有两种可能：

- 不存在虚拟地址到物理页的映射，需要给虚拟地址映射一个物理页
- 存在映射，但物理页被换出到磁盘

练习一主要需要完成第一种情况的处理。首先需要使用lab2中所完成的get_pte获取该虚拟地址的页表项，地址要向下对齐，且是在程序的页表中获得页表项，同时如果这个页表项为0，说明映射不存在，需要分配一页，以建立虚拟地址和物理地址的映射。分配页使用的是pgdir_alloc_page()函数，这个函数会调用alloc_page()和page_insert()为虚拟地址分配新的一页，并建立映射关系，保存在页表当中。

```
    //页表项的权限设置
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);     //对齐
    ret = -E_NO_MEM;                    //error.h中定义的内存请求错误码
    pte_t *ptep=NULL;                   //页表项指针
    ptep = get_pte(mm->pgdir,addr,1);   //获取页表项
    if ( ptep == NULL) {
        cprintf("do_pagefault failed: get_pte failed\n");
        goto failed;
    }
    //映射不存在
    if (*ptep == 0) {
        if(!pgdir_alloc_page(pgdir,addr,perm)){     //建立物理页映射
            cprintf("do_pgfault failed: pgdir_alloc_page failed\n");
        }
        goto failed;
    }
```

在pmm.c中定义的pgdir_alloc_page函数如下：

```
struct Page *
pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page();                   //分配一页
    if (page != NULL) {
        if (page_insert(pgdir, page, la, perm) != 0) {  //插入该页
            free_page(page);
            return NULL;
        }
        if (swap_init_ok){
            swap_map_swappable(check_mm_struct, la, page, 0);
            page->pra_vaddr=la;
            assert(page_ref(page) == 1);
        }
    }
    return page;
}
```

page_insert函数如下：

```
int
page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    pte_t *ptep = get_pte(pgdir, la, 1);
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page);
    /*页表项已经存在的情况需要处理*/
    if (*ptep & PTE_P) {
        struct Page *p = pte2page(*ptep);
        if (p == page) {
            page_ref_dec(page);
        }
        else {
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = page2pa(page) | PTE_P | perm;
    tlb_invalidate(pgdir, la);
    return 0;
}
```

4.问题一

问题一：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。

页目录项和页表项的组成如下：

页目录项的组成

- 前20位表示该PDE对应的页表起始位置
- 第9-11位保留给OS使用
- 第8位可忽略
- 第7位用于设置Page大小，0表示4KB
- 第6位为0
- 第5位表示该页是否被引用过
- 第4位表示是否需要进行缓存
- 第3位表示CPU是否可直接写回内存
- 第2位表示该页是否可被任何特权级访问
- 第1位表示是否允许读写
- 第0位为该PDE的存在位

页表项的组成

- 前20位表示该PTE指向的物理页的物理地址
- 第9-11位保留给OS使用

- 第8位表示在 CR3 寄存器更新时无需刷新 TLB 中关于该页的地址
- 第7位恒为0
- 第6位表示该页是否被写过
- 第5位表示是否被引用过
- 第4位表示是否需要进行缓存
- 第0-3位与页目录项的0-3位相同

页目录项主要是用于找到页表项，在页访问异常处理时，找到页表项后可以判断虚拟地址对应的物理页是否存在，如果不存在需要建立虚拟地址到物理页的映射，存在则说明该页被换出，需要换入。而页表项在页未被换出时，一些位可以用于页替换算法中页的选择，如引用位和修改位在时钟替换算法中会用到，如果页被换出，页表项可以用于保存页被换入硬盘的扇区位置，以便换入。

5.问题二

问题二：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?

在这种情况下发生页访问异常和其他情况下发生页访问异常的处理是相同的。硬件需要将发生错误的线性地址保存到CR2寄存器中，将寄存器以及错误码压入栈中，从中断描述符表中找到并执行缺页服务例程进行处理。

练习二：基于FIFO的页替换算法

1.扩展的Page结构&swap_manager

扩展的Page结构

为了表示页可被换出或已被换出，对保存页信息的Page结构进行了扩展。

```
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status of the page
frame
    unsigned int property;          // the num of free block, used in first fit pm manager
    list_entry_t page_link;         // free list link
    list_entry_t pra_page_link;     // used for pra (page replace algorithm)
    uintptr_t pra_vaddr;            // used for pra (page replace algorithm)
};
```

pra_page_link用来构造按页的第一次引用时间进行排序的一个链表，这个链表的开始表示第一次引用时间最近的页，链表结尾表示第一次引用时间最远的页。pra_vaddr用来记录此物理页对应的虚拟页地址。

swap_manager

为了实现各种页替换算法，ucore使用了页替换算法的类框架swap_manager:

```
struct swap_manager
{
     const char *name;
```

```
     /* swap manager初始化*/
     int (*init)            (void);
     /* mm_struct中的页访问情况初始化 */
     int (*init_mm)         (struct mm_struct *mm);
     /* 时钟中断时调用的函数 */
     int (*tick_event)      (struct mm_struct *mm);
     /* 页访问情况记录 */
     int (*map_swappable)   (struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in);
     int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
     /* 选择需要换出的页 */
     int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page, int in_tick);
     /* 检查页替换算法*/
     int (*check_swap)(void);
};
```

其中最重要的是map_swappable和swap_out_victim，map_swappable将页访问情况更新，swap_out_victim选出需要换出的页，本练习需要完成的是fifo算法实现页替换机制的这两个函数。

2.页的换入

访问的虚拟地址的物理页不存在的第二种情况是存在映射，但物理页被换出到硬盘。这时需要将硬盘上的页换入，这是通过swap_in()函数实现的。swap_in()函数通过传入的地址获取其页表项pte，根据页表项所提供的信息将硬盘上的信息读入，完成页的换入。

```
//swap_in()的定义，完成换入返回0
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result);
//页被换出后，页表项记录了换出的页在硬盘中的位置
//offset表示该页的起始扇区位置
swap_entry_t
-------------------------------------------
|         offset        |   reserved   | 0 |
-------------------------------------------
          24 bits            7 bits    1 bit
```

在处理因页被换出导致的页访问异常时，首先将页换入，设置Page结构的pra_vaddr变量，然后调用page_insert()建立虚拟地址与物理页的映射，由于该页是用户程序使用的，还需要调用swap_map_swappable()将该页在swap_manager 页替换管理框架中设置该页为可替换。以上两个函数的定义如下：

```
//建立虚拟地址与物理页的映射
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm)
//将页设置为可替换
int
swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
```

最终在do_pgfault中完成页换入的实现如下：

```
    if(swap_init_ok) {                  //swap_init中会指定fifo算法实现的页替换类框架
        struct Page *page=NULL;
        int ret=swap_in(mm,addr,&page);             //页的换入
        if(ret!=0){
            cprintf("do_pgfault failed: swap_in failed\n");
```

```
        }
        page->pra_vaddr=addr;
        page_insert(mm->pgdir,page,addr,perm);       //映射的建立
        swap_map_swappable(mm,addr,page,1);          //设置该页为可替换
    }
    else {
        cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
        goto failed;
    }
```

3.map_swappable的实现

在swap_init中，会指定页替换算法框架为swap_manager_fifo。当调用swap_map_swappable更新可替换的页的情况时，会使用页替换算法框架的map_swappable函数。因此需要实现的是_fifo_map_swappable这个函数。该函数的作用就是将访问过的页加入mm_struct结构的sm_priv链表中，以记录该页访问过，且可替换。具体实现只需要将该页的pra_page_link作为节点链接入sm_priv链表中：

```
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);
    return 0;
}
```

4.swap_out_victim的实现

需要实现的是swap_manager_fifo对应的_fifo_swap_out_victim函数，选择出将要换出的页。按照先进先出的算法，需要换出的页是最早被访问的页，这个页在链表的尾部，选出该页后要将该页从sm_priv链表中去除该页，并将该页的地址保存到传入的ptr_page变量中，具体的实现如下：

```
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);
     list entry_t *le=list_prev(head);          //头节点的前一节点即尾节点
     assert(le!=head);
     struct Page *victim=le2page(le);           //找到链表节点对应的页
     list_del(le);                              //删除该页
     assert(victim!=NULL);
     *ptr_page=victim;
     return 0;
}
```

5.问题：extended clock页替换算法实现

换出页的特征

时钟算法是一种近似LRU的算法。时钟指针一开始指向任意的一页，进行页替换时，检查页的引用位为1还是为0，如果为1，则页面最近被引用过，将该页引用位设置为0，继续查找，直到找到引用位为0的页，将该页换出。在此基础之上还可以进行进一步修改，考虑到将已被修改的页换出到磁盘是有代价的，可以在寻找换出页时根据页的修改位判断页是否被修改过，优先选择既没被修改过，又最近没有引用的页优先换出。因此换出页的特征为：最近没有被引问过，没有被修改过。

判断具有换出页特征的页

在页表项的组成中，第六位 PTE_A （Acessed）表示是否被引用过，第五位PTE_D（Dirty）表示该页是否被修改过，根据这两个标志位就可以识别出最近没有被访问过且未修改过的页。

何时进行换入换出操作

当需要访问的页不在内存中时，会引发页访问异常，此时需要进行页的换入，如果发现内存已不足，就需要将页进行换出。

Challenge：识别dirty bit的 extended clock页替换算法

1.swap_manager_extended_clock页替换算法框架

原有的swap_manager足以支持实现extended clock页替换算法。只需要实现一个类似于swap_manager_fifo的swap_manager_extended_clock页替换算法框架就可以实现，而这个框架中的页换入和初始化等函数都可以沿用swap_manager_fifo中的函数实现，需要重新实现的只有选择替换出的页的函数，即需要实现_extended_clock_swap_out_victim函数。因此按照swap_fifo.h和swap_fifo.c文件的格式仿写swap_extended_clock.h和swap_extended_clock.c文件，沿用fifo页替换框架算法中初始化和换入等函数实现，重点完成选择换出页的_extended_clock_swap_out_victim函数，最后在swap.c中的swap_init函数中将使用的页替换算法框架更改为swap_manager_extended_clock，就可以在ucore中使用extended_clock算法实现的页替换了。

2._extended_clock_swap_out_victim的实现

扩展时钟算法的目标是尽可能找到未被引用过且未被写过的页，将该页换出。第一次遍历链表可以直接尝试寻找引用位和修改位都为0的页，如果遍历到的页访问过，将访问位设置为0。如果第一轮没有找到，则进行第二轮的遍历。由于第一轮后所有页的引用位都为0，第二轮可以再次尝试寻找引用位和修改位都为0的页。如果第二轮也没有找到，说明所有页都被修改过了，第三轮直接选择一个引用位为0，修改位为1的页就可以了。具体实现的过程还会用到le2page获取Page结构，get_pte获取页表项，通过页表项进行标志位判断。

最终的实现如下：

```
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);
     list_entry_t *le;
     struct Page *victim;
     for(int i=0;i<2;i++){
        le=list_next(head);
        assert(le!=head);
        while(le!=head){
            victim=le2page(le,pra_page_link);
            pte_t *ptep=get_pte(mm->pgdir,victim->pra_vaddr,0);
            //前两次循环
            if(!(*ptep & PTE_A) && !(*ptep & PTE_D)){
                assert(victim!=NULL);
                *ptr_page=victim;
                list_del(le);
                return 0;
            }
            if(*ptep & PTE_A) *ptep=*ptep & (~PTE_A);
            //修改了标志位，更新TLB
            tlb_invalidate(mm->pgdir, victim->pra_vaddr);
            le=list_next(le);
        }
     }
     //最后直接进行选择,考虑到换入时插入链表的顺序，仍然选择链尾的页
     le=list_prev(head);
     victim=le2page(le,pra_page_link);
     assert(victim!=NULL);
     *ptr_page=victim;
     list_del(le);
     return 0;
}
```

3.测试

结合swap.c和swap_fifo.c中的相关函数与定义，可以判断用于测试的虚拟地址是从0x1000开始，而最多用于测试的有效物理页为4页，在swap.c中会先对0x1000，0x2000，0x3000，0x4000虚拟地址进行四次写操作，这会触发四次页访问异常（因为这些页不在物理内存中），四次建立虚拟地址到物理页的映射，这四个虚拟地址对应的页就在内存中存在了，接下来写0x5000，就会再次产生页访问异常，需要将一页换出，接下来引用刚才被换出的页，页访问异常次数应该加1，这样就可以验证哪一页被换出了，从而验证页替换算法是否正确。

```
//swap.c中与测试相关的定义
// the valid vaddr for check is between 0~CHECK_VALID_VADDR-1
#define CHECK_VALID_VIR_PAGE_NUM 5
#define BEING_CHECK_VALID_VADDR 0X1000
#define CHECK_VALID_VADDR (CHECK_VALID_VIR_PAGE_NUM+1)*0x1000
// the max number of valid physical page for check
#define CHECK_VALID_PHY_PAGE_NUM 4
//测试页替换算法前先使四个页的物理页映射存在
static inline void
check_content_set(void){
     *(unsigned char *)0x1000 = 0x0a;
     assert(pgfault_num==1);
```

```
     *(unsigned char *)0x1010 = 0x0a;
     assert(pgfault_num==1);
     *(unsigned char *)0x2000 = 0x0b;
     assert(pgfault_num==2);
     *(unsigned char *)0x2010 = 0x0b;
     assert(pgfault_num==2);
     *(unsigned char *)0x3000 = 0x0c;
     assert(pgfault_num==3);
     *(unsigned char *)0x3010 = 0x0c;
     assert(pgfault_num==3);
     *(unsigned char *)0x4000 = 0x0d;
     assert(pgfault_num==4);
     *(unsigned char *)0x4010 = 0x0d;
     assert(pgfault_num==4);
}
```

用于测试的页分别标记为页a，b，c，d，e，设计测试时需要对一些页只进行读操作，以验证未被修改过的页会被优先换出：

```
static int
_extended_clock_check_swap(void) {
    int read;
    //写abcd页不会产生异常
    cprintf("write Virt Page c in extended_clock_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==4);
    cprintf("write Virt Page a in extended_clock_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==4);
    cprintf("write Virt Page d in extended_clock_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==4);
    cprintf("write Virt Page b in extended_clock_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==4);
    //读页e，需要换出，此时四个页都被修改过，应该换出最早添加到链表的页a
    cprintf("write Virt Page e in extended_clock_check_swap\n");
    read =*(unsigned char *)0x5000;
    assert(pgfault_num==5);
    //读页a，需要换出，此时e没被修改过，应该换出页e
    cprintf("write Virt Page a in extended_clock_check_swap\n");
    read = 0x0a;
    assert(pgfault_num==6);
    //写页e，需要换出，页访问异常+1，此时a没被修改过，应该换出页a
    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==7);
    //写页a，需要换出，页访问异常+1，此时edcb都写过，应该换出页b
    cprintf("write Virt Page a in extended_clock_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==8);
    return 0;
}
```

测试结果如下：

```
set up init env for check_swap over!
//前四次无页访问异常
write Virt Page c in extended_clock_check_swap
write Virt Page a in extended_clock_check_swap
write Virt Page d in extended_clock_check_swap
write Virt Page b in extended_clock_check_swap
//读页e，换出页a
read Virt Page e in extended_clock_check_swap
page fault at 0x00005000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
//读页a，换出页e
read Virt Page a in extended_clock_check_swap
page fault at 0x00001000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
//写页e，换出页a
write Virt Page e in extended_clock_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
//写页a，换出页b
write Virt Page a in extended_clock_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
count is 0, total is 7
check_swap() succeeded!
...
```

测试结果符合预期，这证明extended clock页替换算法的实现是正确的。

实验总结

与参考答案的对比

练习一

与参考答案基本一致。

练习二

与参考答案基本一致，swap_out_victim中缺少assert(victim!=NULL)确认指针不为空，根据答案补全。

Challenge

仿照FIFO算法实现的页替换算法框架仿写extended clock算法实现的页替换算法框架，并仿照FIFO页替换的测试设计了对应的测试，进行页的替换，最终结果符合预期，未被引用的且未写过的页被优先换出，实现的页替换算法是正确的。

重要知识点

- 页访问异常的处理
- 按需分页
- 先进先出页替换算法
- 时钟算法

本实验主要是关于页访问异常的具体处理，以及页的换入换出，替换算法的具体实现。

未涉及的内容

- 进程管理
- 进程调度
- 文件系统