OS_Lab3_Experimental report

湖南大学信息科学与工程学院

计科 210X 甘晴void （学号 202108010XXX）

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

实验内容

本次实验是在lab2的基础上，借助于页表机制和lab1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lab1和lab2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。

实验概述

本次实验主要完成ucore内核对虚拟内存的管理工作。

总述

首先完成初始化虚拟内存管理机制，即需要设置好哪些页需要放在物理内存中，哪些页不需要放在物理内存中，而是可被换出到硬盘上，并涉及完善建立页表映射、页访问异常处理操作等函数实现。

然后就执行一组访存测试，看看我们建立的页表项是否能够正确完成虚实地址映射，是否正确描述了虚拟内存页在物理内存中还是在硬盘上，是否能够正确把虚拟内存页在物理内存和硬盘之间进行传递，是否正确实现了页面替换算法等。

lab3的总体执行流程

首先是初始化过程。参考ucore总控函数init的代码，可以看到在调用完成虚拟内存初始化的vmm_init函数之前，需要首先调用pmm_init函数完成物理内存的管理，这也是我们lab2已经完成的内容。接着是执行中断和异常相关的初始化工作，即调用pic_init函数和idt_init函数等，这些工作与lab1的中断异常初始化工作的内容是相同的。

在调用完idt_init函数之后，将进一步调用三个lab3中才有的新函数vmm_init、ide_init和swap_init。这三个函数涉及了本次实验中的两个练习。第一个函数vmm_init是检查我们的练习1是否正确实现了。

为了表述不在物理内存中的“合法”虚拟页，需要有数据结构来描述这样的页，为此ucore建立了mm_struct和vma_struct数据结构（后面有进一步详细描述），假定我们已经描述好了这样的“合法”虚拟页，当ucore访问这些“合法”虚拟页时，会由于没有虚实地址映射而产生页访问异常。如果我们正确实现了练习1，则do_pgfault函数会申请一个空闲物理页，并建立好虚实映射关系，从而使得这样的“合法”虚拟页有实际的物理页帧对应。这样练习1就算完成了。

ide_init和swap_init是为练习2准备的。由于页面置换算法的实现存在对硬盘数据块的读写，所以ide_init就是完成对用于页换入换出的硬盘（简称swap硬盘）的初始化工作。完成ide_init函数后，ucore就可以对这个swap硬盘进行读写操作了。swap_init函数首先建立swap_manager，swap_manager是完成页面替换过程的主要功能模块，其中包含了页面置换算法的实现（具体内容可参考5小节）。然后会进一步调用执行check_swap函数在内核中分配一些页，模拟对这些页的访问，这会产生页访问异常。如果我们正确实现了练习2，就可通过do_pgfault来调用swap_map_swappable函数来查询这些页的访问情况并间接调用实现页面置换算法的相关函数，通过某种算法（FIFO或是时钟算法等）把“不常用”的页换出到磁盘上。

ucore在实现上述技术时，需要解决三个关键问题：

1. 当程序运行中访问内存产生page fault异常时，如何判定这个引起异常的虚拟地址内存访问是越界、写只读页的“非法地址”访问还是由于数据被临时换出到磁盘上或还没有分配内存的“合法地址”访问？
2. 何时进行请求调页/页换入换出处理？
3. 如何在现有ucore的基础上实现页替换算法？

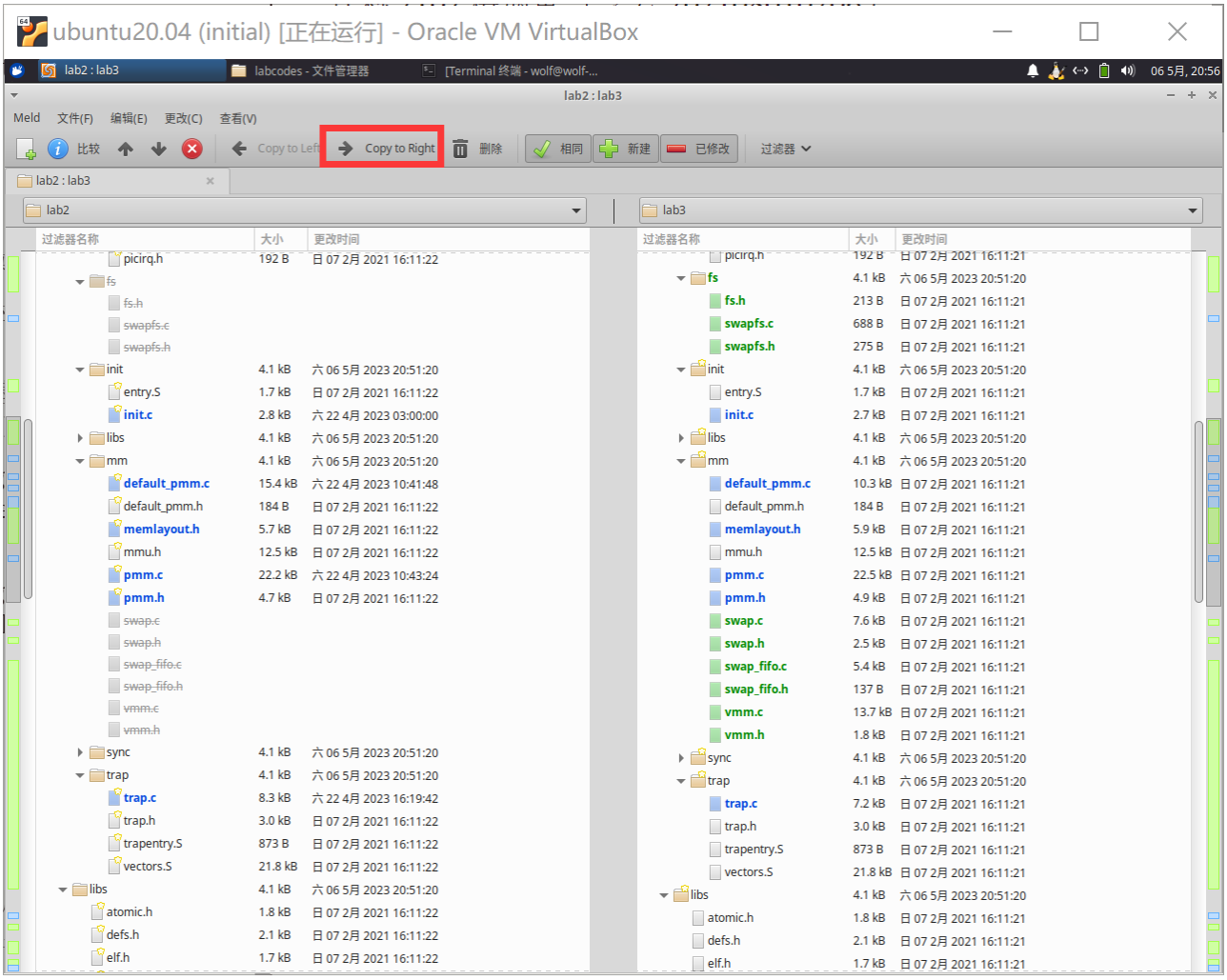
接下来将进一步分析完成lab3主要注意的关键问题和涉及的关键数据结构。

练习0：填写已有实验

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。

使用`sudo apt install meld`安装meld工具

使用meld工具可以比较方便地查看Lab2与Lab3的差异，由于Lab1已经是被Lab2兼容了，所以不需要再做考虑。



如图所示，在关键代码部分点击红框按键，可以完成填补工作。注意不要对其他代码进行更新，以免造成错误。

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。注意：在LAB3 EXERCISE 1处填写代码。执行`make qemu`后，如果通过check_pgfault函数的测试后，会有“check_pgfault() succeeded!”的输出，表示练习1基本正确。

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：

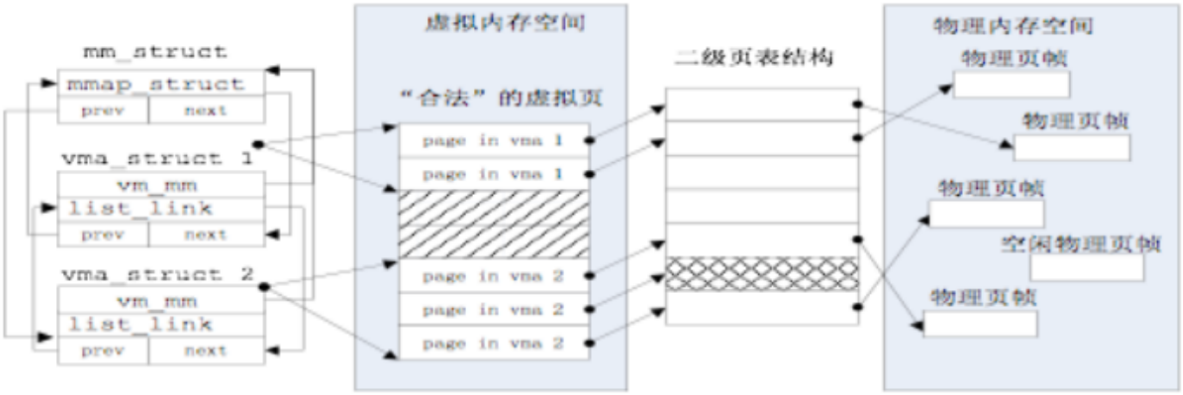
- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处。
- 如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情？

1.页访问异常

（1）有关虚拟内存的注意点

- 虚拟内存单元不一定有实际的物理内存单元对应，即实际的物理内存单元可能不存在；
- 如果虚拟内存单元对应有实际的物理内存单元，那二者的地址一般是不相等的；
- 通过操作系统实现的某种内存映射可建立虚拟内存与物理内存的对应关系，使得程序员或CPU访问的虚拟内存地址会自动转换为一个物理内存地址。

（2）虚拟地址空间和物理地址空间



一块虚拟地址（vma）可能映射到一个或多个多个物理页。由于一开始，分配虚拟空间超过了物理空间的大小，比如在上图中只有5个物理页帧，但其实我们给它分了7个虚拟页，那很明显一定会有两个虚拟页没有对应的物理页帧。那么如果访问到这两个虚拟页，在二级页表里面它会没有对应映射关系，一旦没有对应映射关系，那就会产生缺页异常，因此就是do_pgfault函数的实现功能，它要建立这个映射关系。

（3）页访问异常类型

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)，此时应该直接报错；
- 相应的物理页帧不在内存中（页表项非空，但Present标志位=0)，此时应该建立映射虚拟页和物理页的映射关系；
- 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)，此时应该直接报错.

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*

（4）从**lab2**出发的更进一步

lab2中有关内存的数据结构和相关操作都是直接针对实际存在的资源–物理内存空间的管理，没有从一般应用程序对内存的“需求”考虑，即需要有相关的数据结构和操作来体现一般应用程序对虚拟内存的“需求”。一般应用程序的对虚拟内存的“需求”与物理内存空间的“供给”没有直接的对应关系，ucore是通过page fault异常处理来间接完成这二者之间的衔接。

2.数据结构mm_struct和vma_struct

page_fault函数不知道哪些是“合法”的虚拟页，原因是ucore还缺少一定的数据结构来描述这种不在物理内存中的“合法”虚拟页。为此ucore通过建立mm_struct和vma_struct数据结构，描述了ucore模拟应用程序运行所需的合法内存空间。

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。

（1）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的操作函数只有如下的两个函数：

- mm_create
- mm_destroy

分别用于创建该变量和删除变量并释放空间。

（2）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

下面是对它们的解释

- 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);
```

（1）函数参数解析

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)

（2）前置步骤（虚拟地址存在性与读写权限合规性）

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。

```
//★这一段代码主要排除虚拟地址范围超过限制或者无法被查找到的问题
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：

```
//★这一段代码主要排除读写权限不符合的问题
//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。

★（3）实现do_pgfault（mm/vmm.c）函数

```
uint32_t perm = PTE_U;//prem:给物理页赋予权限的中间变量
    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
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {//获取页表项
        cprintf("do_pagefault failed: get_pte failed\n");//没有对应的二级页表项，它根本不存在，不知道用什么物理页去映射，报错。注意到【这里不可能不存在，如果查找到不存在的情况，由于get_pte的create标记位为1，那么会创建一个新的二级页表】
        goto failed;
    }
    //映射不存在
    if (*ptep == 0) {//如果是上述新创建的二级页表，那么*ptep就会是0，代表页表为空，调用pgdir_alloc_page，对它进行初始化
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
```

```
            cprintf("do_pgfault failed: pgdir_alloc_page
failed\n");
            goto failed;
        }
    }
```

接下来放的是在pmm.c中定义的这两个执行函数

（4）pgdir_alloc_page

```
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;
}
```

（5）page_insert函数

```
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;
}
```

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。而页表项可以记录该虚拟页在磁盘中的位置，为换入换出提供磁盘位置信息。页目录项则是用来索引对应的页表。

分页机制的实现，确保了虚拟地址和物理地址之间的对应关系，一方面，通过查找虚拟地址是否存在于一二级页表中，可以容易发现该地址是否是合法的，另一方面，通过修改映射关系即可实现页替换操作。另外，基于页表实现了地址的分段操作，在这里，一个物理地址不同的位数上，会存储一系列不同的信息，例如物理页是否存在，是否可读，对应的物理页用户态是否可以访问。

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？

- 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值（简称页访问异常错误码，errorCode）保存在中断栈中。之后通过上述分析的trap–> trap_dispatch–>pgfault_handler–>do_pgfault调用关系，一步步做出处理。

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_victim函数。通过对swap的测试。注意：在LAB3 EXERCISE 2处填写代码。执行`make qemu`后，如果通过check_swap函数的测试后，会有“check_swap() succeeded!”的输出，表示练习2基本正确。

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。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？

★1.与练习1的关联

在练习2中，当页错误异常发生时，有可能是因为页面保存在swap区或者磁盘文件上造成的，所以我们需要利用页面替换算法解决这个问题。
而练习1和这里的关联就在于：页面替换主要分为两个方面，页面换出和页面换入。练习1实现的是页面换入，主要在上述的do_pgfault()函数实现；而练习2这里主要实现，页面换出，主要是在swap_out_vistim()函数。另外，在练习2还有一个函数叫做swappable，代表将该页面设置为可交换的。于是，练习2主要是对于这两个函数的实现。

2.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。

3.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：

在上一练习的原有if后加上这一层else，表示处理的是第二种情况

```
//处理存在映射，但物理页被换出到磁盘的情况
else { // if this pte is a swap entry, then load data from disk to
a page with phy addr
           // and call page_insert to map the phy addr with logical
addr
        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;
            if ((ret = swap_in(mm, addr, &page)) != 0) {//页的换入
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            page_insert(mm->pgdir, page, addr, perm);
            swap_map_swappable(mm, addr, page, 1);//映射的建立
            page->pra_vaddr = addr;//设置该页为可替换
        }
        else {
            cprintf("no swap_init_ok but ptep is %x,
failed\n",*ptep);
            goto failed;
        }
   }
```

4.★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;
}
```

5.★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;
}
```

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

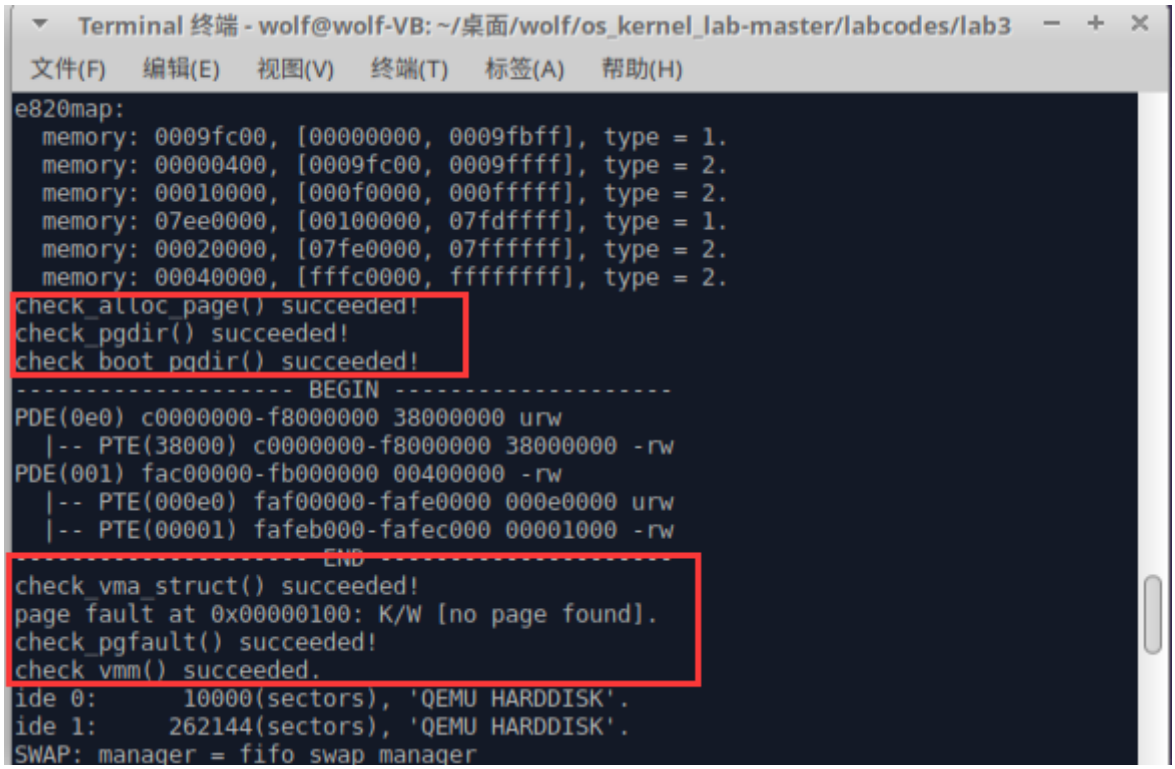
- 当需要访问的页不在内存中时，会引发页访问异常，此时需要进行页的换入；
- 如果发现内存已不足，就需要将页进行换出。

★6、演示

使用make qemu进行模拟演示

```
wolf@wolf-VB:~/桌面/wolf/os_kernel_lab-master/labcodes/lab3$ make qemu
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed
raw.
         Automatically detecting the format is dangerous for raw images, write
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'bin/swap.img' and probing guessed
aw.
         Automatically detecting the format is dangerous for raw images, write
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc0100036 (phys)
  etext  0xc0108539 (phys)
  edata  0xc0125000 (phys)
  end    0xc0126110 (phys)
Kernel executable memory footprint: 153KB
ebp:0xc0121f48 eip:0xc0100ad4 args:0x00010094 0x00010094 0xc0121f78 0xc01000c8
    kern/debug/kdebug.c:308: print_stackframe+25
ebp:0xc0121f58 eip:0xc0100de8 args:0x00000000 0x00000000 0x00000000 0xc0121fc8
```



发现出现succeeded等字样，表示实验成功。

扩展练习 **Challenge 1**：实现识别**dirty bit**的 **extended clock**页替换算法（需要编程）

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。

总结来说，重点在于_extended_clock_swap_out_victim()函数的实现，其他的部分沿用FIFO的就可以（细节上要注意改变）

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：

★swap_extended_clock.h文件

```
//swap_extended_clock.h文件（参照swap_fifo.h的格式修改）
#ifndef __KERN_MM_SWAP_EXTENDED_CLOCK_H__
#define __KERN_MM_SWAP_EXTENDED_CLOCK_H__
#include <swap.h>
extern struct swap_manager swap_manager_clock;
#endif
```

★swap_extended_clock.c文件

```
//swap_extended_clock.c文件
//重点是static int_clock_swap_out_victim函数的实现，其它的依照格式修改即可
#include <defs.h>
#include <x86.h>
#include <stdio.h>
#include <string.h>
#include <swap.h>
```

```c
#include <swap_extended_clock.h>//这个头文件一定注意改成clock的而不是fifo
的
#include <list.h>

list_entry_t pra_list_head;

static int
_clock_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     return 0;
}
//此处和FIFO的初始化方法相同

static int
_clock_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);// 新插入的页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;
}

//这个函数的实现是重点
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);

     list_entry_t *p = head;
     while (1) {
```

```
            p = list_next(p);
            if (p == head) {
                p = list_next(p);
            }
            struct Page *ptr = le2page(p, pra_page_link);
            pte_t *pte = get_pte(mm -> pgdir, ptr -> pra_vaddr, 0);
            //获取页表项
            if ((*pte & PTE_D) == 1) {// 如果dirty bit为1，改为0
                *pte &= ~PTE_D;
            }
            else
            {// 如果dirty bit为0，则标记为换出页
                *ptr_page = ptr;
                list_del(p);
                break;
            }
        }
        return 0;
}

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

```
    assert(pgfault_num==6);
    cprintf("write Virt Page b in clock_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==7);
    cprintf("write Virt Page c in clock_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==8);
    cprintf("write Virt Page d in clock_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==9);
    cprintf("write Virt Page e in clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==10);
    cprintf("write Virt Page a in clock_check_swap\n");
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==11);
    return 0;
}

static int
_clock_init(void)
{
    return 0;
}

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

static int
_clock_tick_event(struct mm_struct *mm)
{ return 0; }

struct swap_manager swap_manager_clock =
{
     .name            = "extend_clock swap manager",
     .init            = &_clock_init,
     .init_mm         = &_clock_init_mm,
     .tick_event      = &_clock_tick_event,
```

```
    .map_swappable    = &_clock_map_swappable,
    .set_unswappable = &_clock_set_unswappable,
    .swap_out_victim = &_clock_swap_out_victim,
    .check_swap       = &_clock_check_swap,
};
```

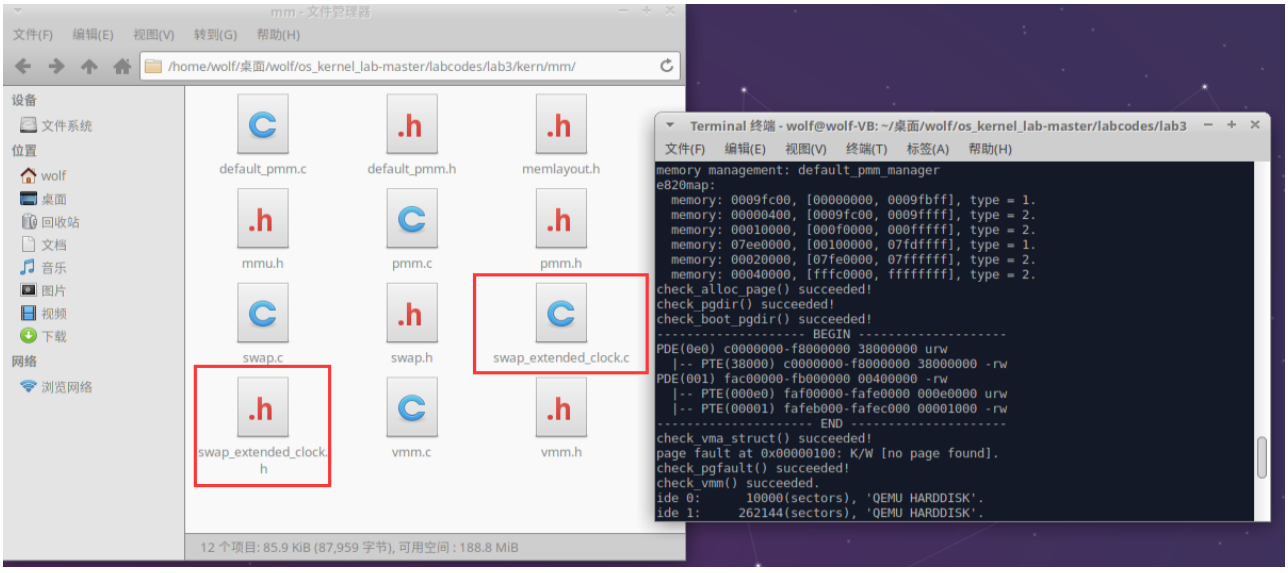
★swap.c文件修改部分

```
//swap.c文件修改部分
sm = &swap_manager_clock;
```

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。

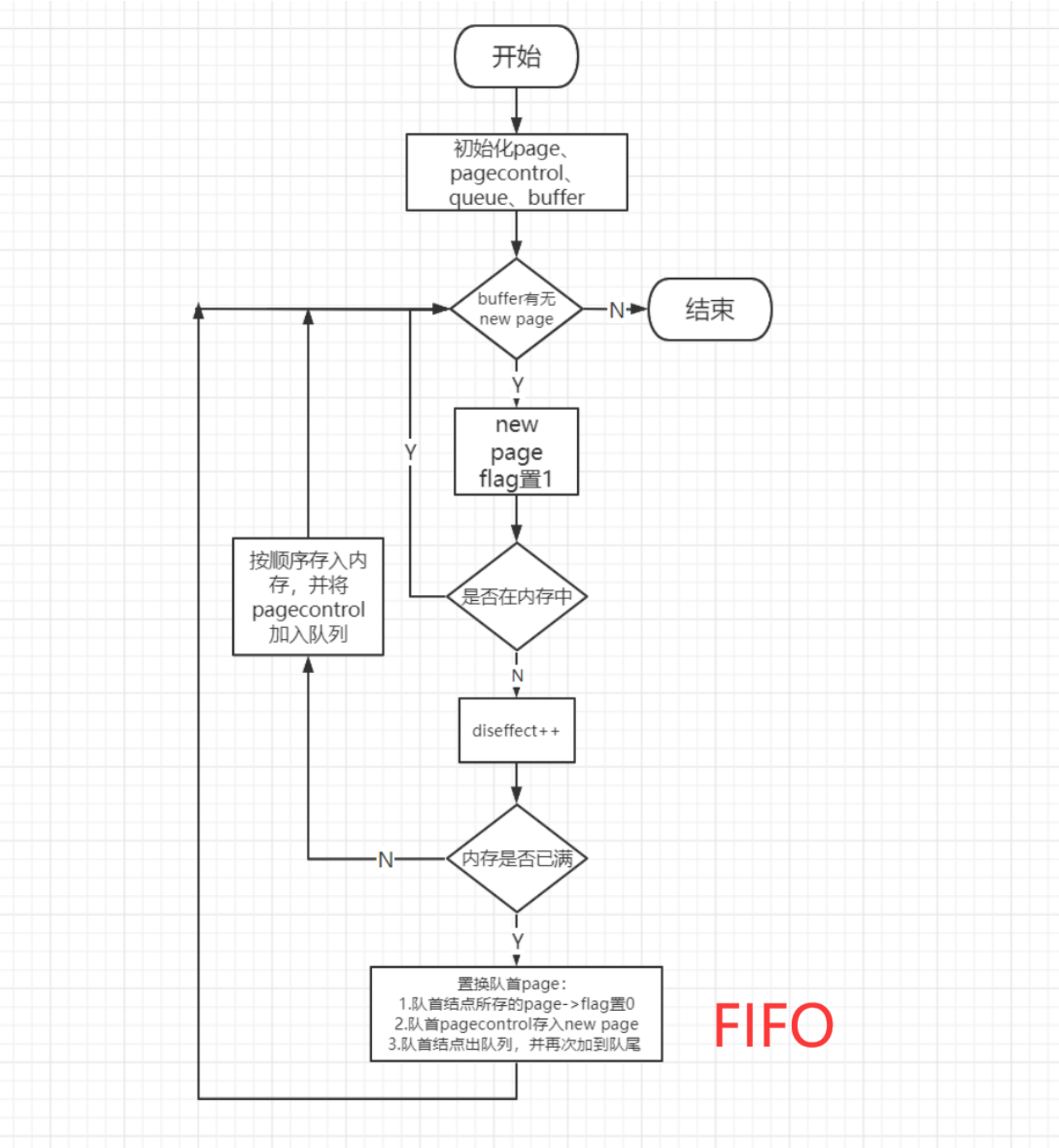
★千万注意要将原有的swap_fifo移走不放在同一个文件夹内，并且注意将swap.c中swap_init函数做修改：即sm = &swap_manager_fifo 改成 sm = &swap_manager_clock。

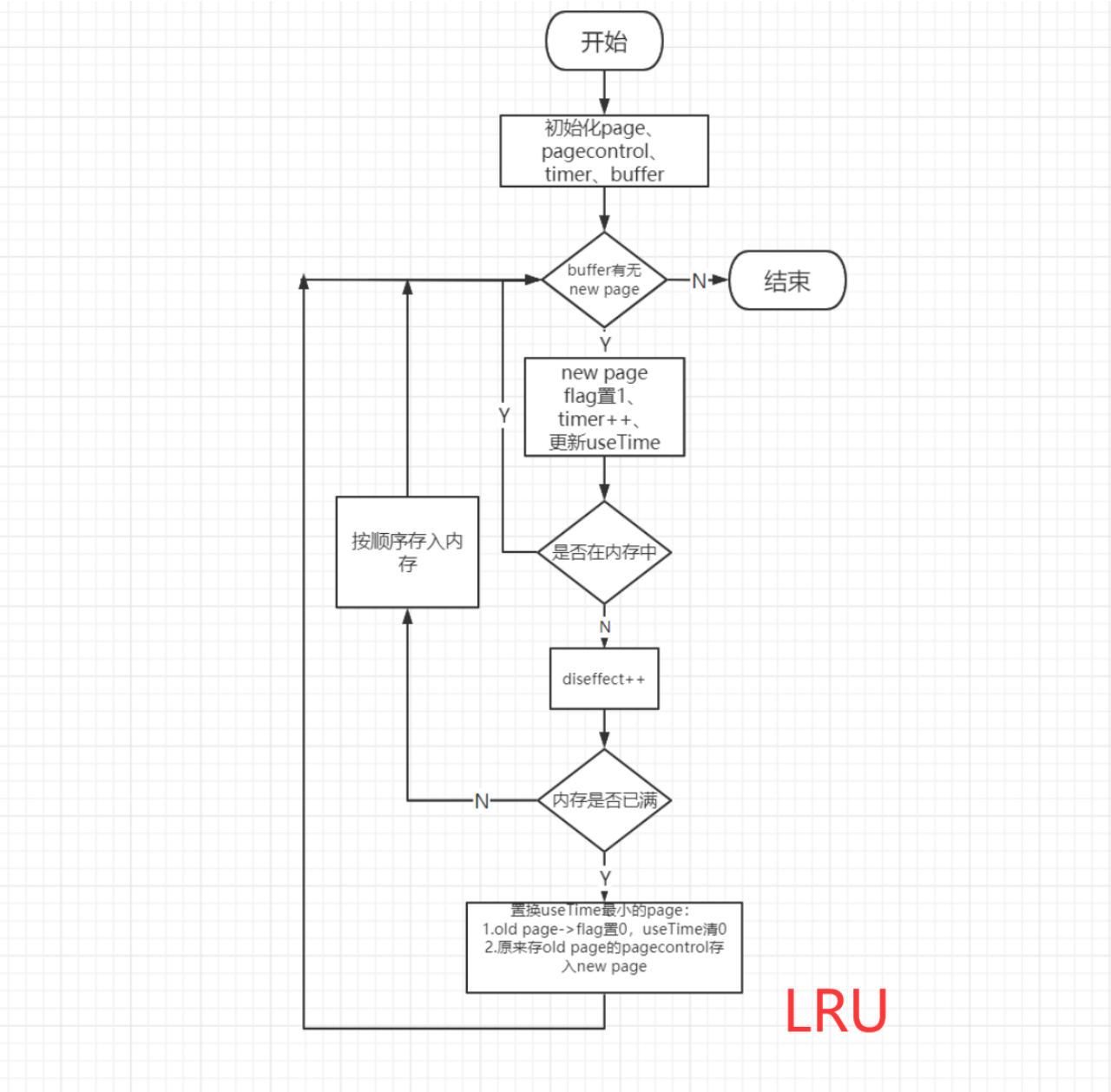


测试结果符合预期，这证明extended clock页替换算法的实现是正确的。

扩展练习 **Challenge 2**：实现不考虑实现开销和效率的**LRU**页替换算法（需要编程）

1.FIFO与LRU的区别





FIFO算法的原理就是将最先进来的给替换掉，不考虑进程频繁切换进程的后果。

LRU（LeastRecentlyUsed）算法的原理是将最近一段时间内没有访问到的进程给替换掉，也就是LRU只考虑最久没有被访问的页面将其替换掉。

Fifo算法的缺点很显而易见就是因为它总是优先把先进来的替换掉，如果在一个程序里面交替出现了很多次这样的页面，那么这个页面会不断地把前面的项替掉因此会出现很多缺页，而lru算法就可以避免这个问题。

FIFO算法相对比较简单，因为只需要设置一个指针，每次往后指就好了，而写LRU算法的时候计算每个进程的等待时间是一个挺麻烦的事情，一个不小心就导致全部出错。

2.实现算法

计数器实现法：在最简单的情况下，为每个页表条目关联一个使用时间域，并为CPU添加一个逻辑时钟或计数器。每次内存引用都会递增时钟；且每当进行页面引用时，时钟寄存器的内容会复制到相应页面的页表条目的使用时间域。这样我们总是找到每个页面最后被引用的时间，便可以置换拥有最小使用时间的页面。

3、具体实现代码

具体实现代码如下，尚未整合至系统中

```
#include<bits/stdc++.h>
using namespace std;
class LinkStack{
private:
    struct Node{
        int elem;
        struct Node* next;
        struct Node* prev;
    };
    struct Node* head;
    struct Node* tail;
public:
    LinkStack(){
        head=new struct Node;
        tail=new struct Node;
        head->next=NULL;
        head->prev=NULL;
        tail=head;
    }
    ~LinkStack(){}
    void push(int k){
        struct Node* middle = new struct Node;
        middle->elem=k;
        tail->next=middle;
        middle->prev=tail;
        middle->next=NULL;
        tail=middle;
    }
    void pop(){
        if(tail==head){
            cout<<"the stack is empty";
```

```
        }
        else{
            struct Node* middle=new struct Node;
            middle=tail;
            tail=tail->prev;
            //tail->prev->next=tail;
            tail->next=NULL;
            free(middle);
        }
    }
    void del(int k){
        struct Node* ser = new struct Node;
        ser=head;
        while(ser=ser->next){
            if(ser->elem==k){
                ser->prev->next=ser->next;
                ser->next->prev=ser->prev;
                tail->next=ser;
                ser->prev=tail;
                ser->next=NULL;
                tail=ser;
                break;
            }

        }

    }
    bool findlink(int k){
        struct Node* ser = new struct Node;
        ser=head;
        while(ser=ser->next){
            if(ser->elem==k){
                return true;
            }
        }
        return false;
    }
    void print_structure(){
        struct Node* ser=new struct Node;
        ser=tail;
        while(ser!=head){
```

```
            cout<<"+--------------+\n";
            cout<<ser->elem<<endl;
            cout<<"+--------------+\n";
            ser=ser->prev;
        }
    }
    int feedback(int k){
        int level=1;
        struct Node* ser=new struct Node;
        ser=head;
        while(ser=ser->next){
            if(ser->elem==k){
                return level;
            }
            level++;
        }
    }
    int back_head(){
        return head->next->elem;
    }
};
int main()
{
    int k;LinkStack s;
    cout<<"请输入空闲的帧的数目:\n";
    int p;cin>>p;int sum=p;
    int element[p];
    while(1==1){
          cout<<"以下输入一个引用串:"<<endl;
          cin>>k;
          if(p>0){
            cout<<"具有空闲帧\n";
            int flag=1;
            for(int i=0;i<sum-p;i++){
                if(element[i]==k){
                    cout<<"当前引用串已经存在于页帧中，故无需请求调页\n";
                    flag=0;break;
                }
            }
            if(flag==1){
                for(int i=0;i<sum-p;i++){
                    cout<<element[i]<<endl;
```

```
                }
                element[sum-p]=k;
                cout<<k<<endl;
                for(int i=sum-p+1;i<sum;i++){
                    cout<<"空闲"<<endl;
                }
                p--;
            }
        }
        else if(p==0){
            int flag=0;
            for(int i=0;i<sum;i++){
                if(element[i]==k){
                    cout<<"该引用已经存在于页帧中，故无需进行页面置换\n";
                    flag=1;break;
                }
            }
            if(flag==0){
                int mind=1e6+10,j=0;
                cout<<"没有空闲帧且该引用不存在于当前的页帧中，开始进行页面置
换\n";
                for(int i=0;i<sum;i++){
                    if(s.feedback(element[i])<mind){
                        j=i;mind=s.feedback(element[i]);
                    }
                }
                element[j]=k;
                for(int i=0;i<sum;i++){
                    cout<<element[i]<<endl;
                }
            }
            if(!s.findlink(k)){
                s.push(k);
            }
            else{
                s.del(k);
            }
        }
    }
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
/*实现方法与FIFO算法大同小异，无需定义队列，而是用计数器实现选择牺牲帧*/
#define ProcessPageNum 10        //进程页面数
#define MemPageNum 5             //内存页面数
#define totalInstruction 100     //引用次数
int diseffect = 0;               //缺页错误数
/*定义进程页面结构体*/
struct pageNode{
    int pageID;              //进程页面ID
    int flag;                //1表示页面在内存中，0表示不在
    int pagecontrolIndex;    //所在内存页面ID 不在内存中时值为-1
    int useTime;             //最后被引用时间
};
/*定义内存页面结点*/
struct pagecontrolNode{
    int pagecontrolID;       //内存页面ID
    int flag;                //1表示被占用，0表示空闲
    int pageIndex;           //所加载进程页面ID
};
/*初始化page数组*/
void initPage(struct pageNode *page[ProcessPageNum]){
    int i = 0;
    for(i;i < ProcessPageNum;i++){
        page[i] = (struct pageNode*)malloc(sizeof(struct
pageNode));
        page[i]->flag = 0;
        page[i]->pageID = i;
        page[i]->pagecontrolIndex = -1;
        page[i]->useTime = 0;
    }
}
/*初始化pagecontrol数组*/
void initPagecontrol(struct pagecontrolNode
*pagecontrol[MemPageNum]){
    int i = 0;
    for(i;i < MemPageNum;i++){
        pagecontrol[i] = (struct
pagecontrolNode*)malloc(sizeof(struct pagecontrolNode));
        pagecontrol[i]->pagecontrolID = i;
        pagecontrol[i]->flag = 0;
```

```
        pagecontrol[i]->pageIndex = -1;
    }
}
/*判断内存页面是否被占满 1为满*/
int isFull(struct pagecontrolNode *pagecontrol[MemPageNum]){
    int i = 0;
    for (i;i < MemPageNum;i++){
        if(pagecontrol[i]->flag == 0) return 0;
    }
    return 1;
}
/*找到内存中页面最后引用时间最小的page结点并返回*/
struct pageNode* findMinUseTime(struct pageNode
*page[ProcessPageNum]){
    struct pageNode *temp;
    temp = (struct pageNode*)malloc(sizeof(struct pageNode));
    temp->useTime = totalInstruction;
    int i = 0;
    for(i;i < ProcessPageNum;i++){
        if(page[i]->useTime < temp->useTime && page[i]->flag == 1){
            temp = page[i];
        }
    }
    return temp;
}
int main(){
    struct pageNode *page[ProcessPageNum];
    struct pagecontrolNode *pagecontrol[MemPageNum];
    initPage(page);
    initPagecontrol(pagecontrol);
    /*构造随机序列buffer*/
    int buffer[totalInstruction];
    int i = 0;
    srand((unsigned)time(NULL));
    for(i;i < totalInstruction;i++){
        buffer[i] = rand() % ProcessPageNum;
    }
    int j = 0;
    int timer = 0;  //设置计数器
    int index = 0;  //当前内存页面下标
    for(j;j < totalInstruction;j++){
        timer++;    //计数器递增
```

```
            page[buffer[j]]->useTime = timer;
            if(page[buffer[j]]->flag == 0){ //若当前页面不在内存中
                diseffect++;                //缺页错误数加1
                page[buffer[j]]->flag = 1;
                if(isFull(pagecontrol)){    //若内存已满
                    struct pageNode *temp = findMinUseTime(page);   //
找到将被置换的页面
                    temp->flag = 0;
                    temp->useTime = 0;
                    pagecontrol[temp->pagecontrolIndex]->pageIndex =
page[buffer[j]]->pageID;   //修改pagecontrol数组
                    page[buffer[j]]->pagecontrolIndex =
pagecontrol[temp->pagecontrolIndex]->pagecontrolID; //修改page数组
                    temp->pagecontrolIndex = -1;
                }else{                      //若内存未满，则按顺序存入
                    pagecontrol[index]->flag = 1;
                    pagecontrol[index]->pageIndex = page[buffer[j]]-
>pageID;
                    page[buffer[j]]->pagecontrolIndex =
pagecontrol[index]->pagecontrolID;
                    index++;
                }
            }
        }
        double rightRate = (1.0 - ((double)diseffect /
(double)totalInstruction))*100;
        printf("错误次数%d\n",diseffect);
        printf("正确率%.2f%%\n",rightRate);
}
```

参考答案对比

练习1

- 表达上有些繁琐，其余基本相似。

练习2

- 与参考答案相比，使用的临时变量较多，不是十分简洁，其余基本相似。

重要知识点和对应原理

实验中的重要知识点

- 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

对应的OS原理知识点

- 页访问异常的具体处理
- 页的换入换出
- 替换算法的具体实现

二者关系

- 本实验设计的知识是对OS原理的具体实现，在细节上非常复杂。

未对应的知识点

- 暂无

参考文献

对于报告主体的流程与实验的进行参考了

- https://blog.csdn.net/Aaron503/article/details/130301784?spm=1001.2014.3001.5501
- https://www.codenong.com/cs110710114/
- https://blog.csdn.net/sfadjlha/article/details/124716093?spm=1001.2014.3001.5502

对于challenge2的思考与代码实现参考了

- https://blog.csdn.net/zhashuang/article/details/127659697