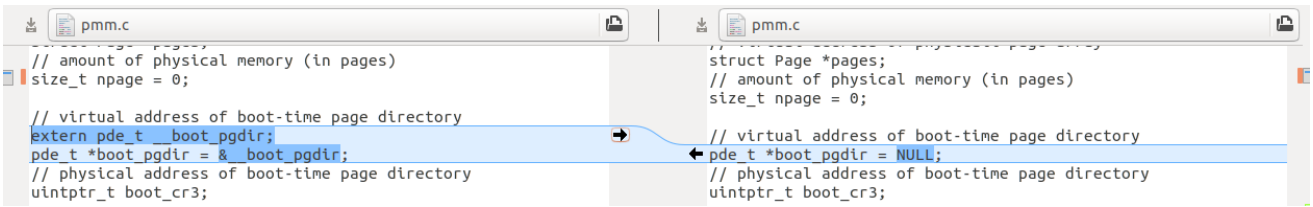
实验三：虚拟内存管理

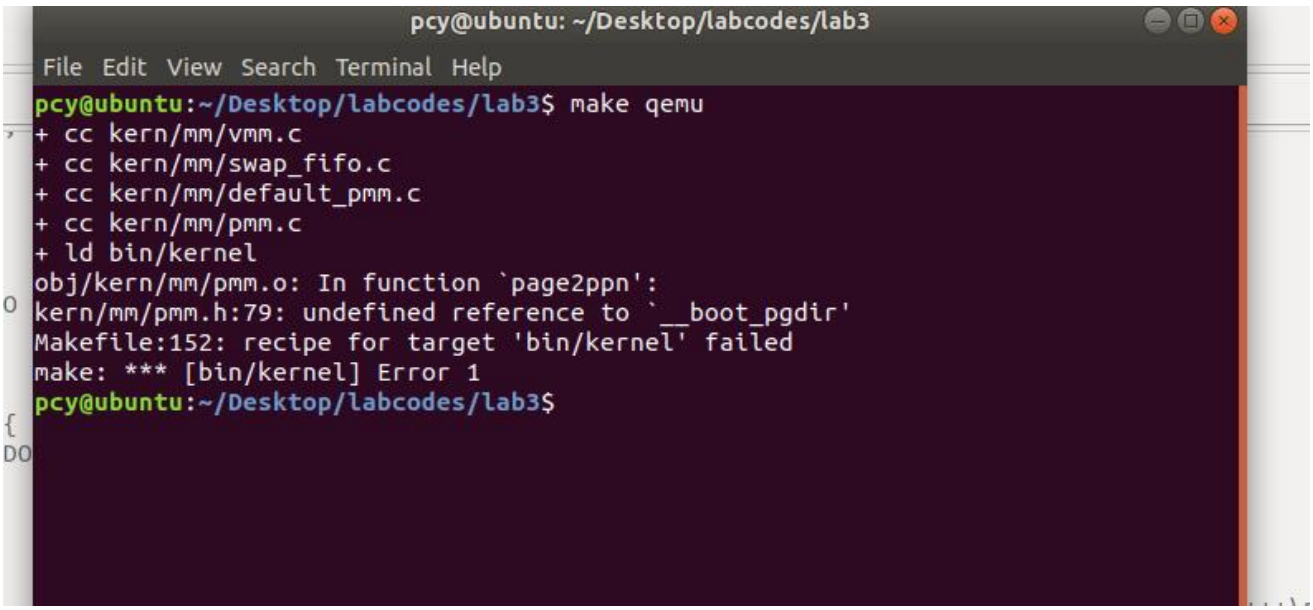
在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成Page Fault异常处理和FIFO页替换算法的实现。设计如何在磁盘上缓存内存页，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。

实验中遇到的问题

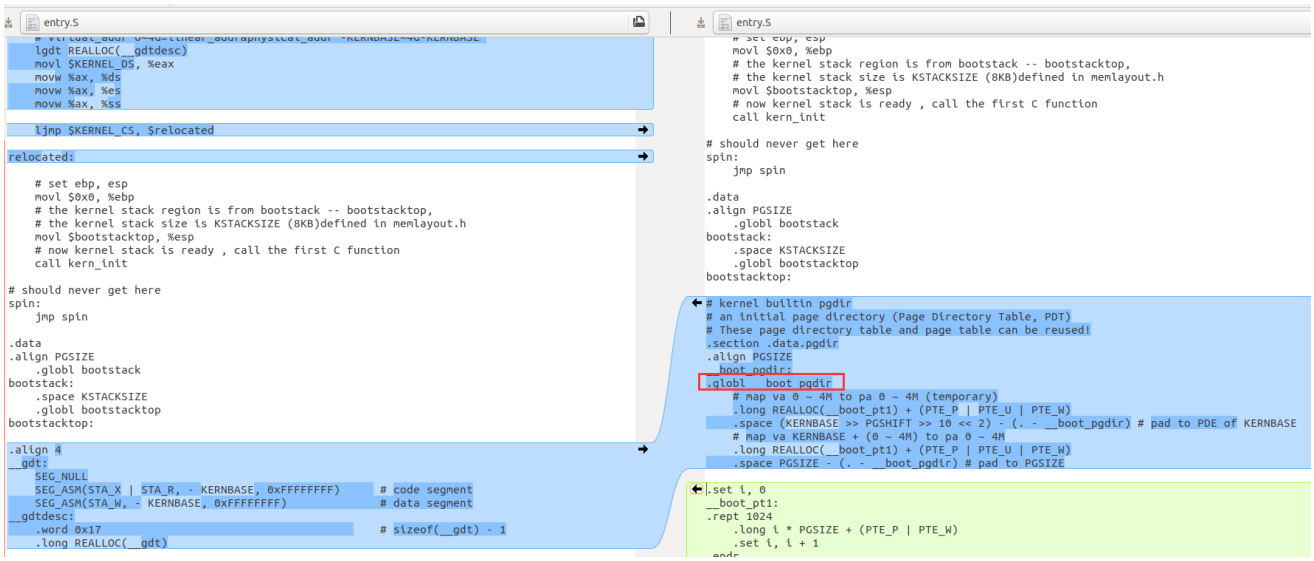
在合并lab1/2/3时，在pmm.c中，对boot_pgdir的赋值不同，理论上这块儿是没有经过改动的，不知道是不是因为代码的版本不一样，所以出现差别。



选用了左边的代码之后，执行make qemu，报错：



查了一下，__boot_pgdir是定义在kern/init/entry.S中的全局变量，entry.S中创建了一个临时的页目录__boot_pgdir 和页表 __boot_pt1，将0~4m和KERNBASE~KERNBASE+4m的虚拟地址映射到0~4m的物理地址上。但是在原来的lab3的entry.S中没有定义__boot_pgdir，所以报错了。但对于最开始那一个版本中为什么可以把boot_pgdir设置为NULL，我还是没有特别清楚。



这次实验大概算是前三次里面最顺利的一次了，所以可写的地方也不是很多。不过上面那个boot_pgdir变量还是不太清楚。下面是平淡无奇的实验过程记录。challenge部分是我从网上找的代码，因为写操作系统的代码对于我来说还是有点艰难，不过我尽力把别人的代码看懂了。

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

练习

练习0：填写已有实验

使用meld快速合并。

需要合并的文件：kern/debug/kdebug.c、kern/mm/pmm.c、kern/trap/trap.c

练习1：给未被映射的地址映射上物理页

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。

```
/* MACROs or Functions:
*get_pte : get an pte and return the kernel virtual address of this pte for la if the
PT contians this pte didn't exist, alloc a page for PT (notice the 3th parameter '1')
*pgdir_alloc_page : call alloc_page & page_insert functions to allocate a page size
memory & setup an addr map pa<--->la with linear address la and the PDT pgdir
*/
/*LAB3 EXERCISE 1: YOUR CODE*/
ptep = get_pte(mm->pgdir,addr,1);//(1) 根据引发缺页异常的地址 去找到 地址所对应的 PTE 如果找不到 则创建一页表
if (*ptep == 0) {//(2) PTE 所指向的 物理页表地址 若不存在 则分配一物理页并将逻辑地址和物理地址作映射 (就是让 PTE 指向 物理页帧)
    if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
        goto failed;
    }
}
```

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。

1. AVL CPU 不理会这个属性 可以不管 (有可能在32位系统使用大过 4G内存的时候 用到这几位)。
2. G Global 全局位 表示是否将虚拟地址与物理地址的转换结果缓存到 TLB 中。
3. D Dirty 脏页位 当 CPU 对这个页进行写操作时 会置 1。
4. PAT Page Attribute Table 页属性表位 置 0。
5. A Accessed 访问位 若为 1 则 说明 CPU 访问过了。CPU 会定时清 0，记录被置 1 的频率，当内存不足时 会将使用频率较低的页面换出到外存 同时将 P位 置 0，下次访问 该页时 会引起 Pagefault 异常 中断处理程序再将此页换上。
6. PCD Page-level Cache Disable 页级高速缓存位，置 0 即可，读的时候 高速缓存是否有效 若有效则直接从高速缓存中读出 若无效的话 则必须从 I/O 端口去读数据。
7. PWT Page-level Write-Through 页级通写位，控制是先写到高速缓存里再慢慢回写到内存里 还是 直接慢慢写到内存里。
8. US User/Superviosr 普通用户/超级用户位
9. RW Read/Write 读写位
10. P Present 存在位 (虚拟页式存储的关键位 若为 0 则发起缺页异常)

- 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?

  页访问异常 会将产生页访问异常的线性地址存入 cr2 寄存器中，并且给出 错误码 error_code ，说明是页访问异常的具体原因。 uCore OS 会将其 存入 struct trapframe 中 tf_err， 等到中断服务例程 调用页访问异常处理函数(do_pgfault()) 时， 再判断 具体原因 。若不在某个VMA的地址范围内 或 不满足正确的读写权限，则是非法访问。 若在此范围 且 权限也正确， 则 认为是 合法访问 ，只是没有建立虚实对应关系 ，应分配一页 并修改页表。 完成 虚拟地址到 物理地址的映射， 刷新 TLB ，最后再 调用 iret 重新执行引发页访问异常的那条指令 。若是在外存中 则将其换入 内存， 刷新 TLB 然后退出中断服务例程， 重新执行引发页访问异常的那条指令。

练习2：补充完成基于FIFO的页面替换算法

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。

```
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);
    //record the page access situlation
    /*LAB3 EXERCISE 2: YOUR CODE*/
    list_add(head, entry); // (1) 将这一页加入到链表头中(最近访问过的放前面) 使其可以被置换算法使
用到
    return 0;
}
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);
     /* Select the victim */
     /*LAB3 EXERCISE 2: YOUR CODE*/
     list_entry_t *le = head->prev;//(1) 换出最先进来的页 (因为每次访问一个页 都是插入到头节点的后面 因此 头节点的前面就是最先访问的页)
     struct Page* page = le2page(le, pra_page_link); // 通过 le 这个链表节点的地址 减去 pra_page_link 在 Page 结构体中的 Offset 得到 Page 的地址
     list_del(le); // 删掉这个节点
     *ptr_page = page; //(2) 将这一页地址存到 ptr_page 中 给 调用本函数的函数使用
     return 0;
}
```

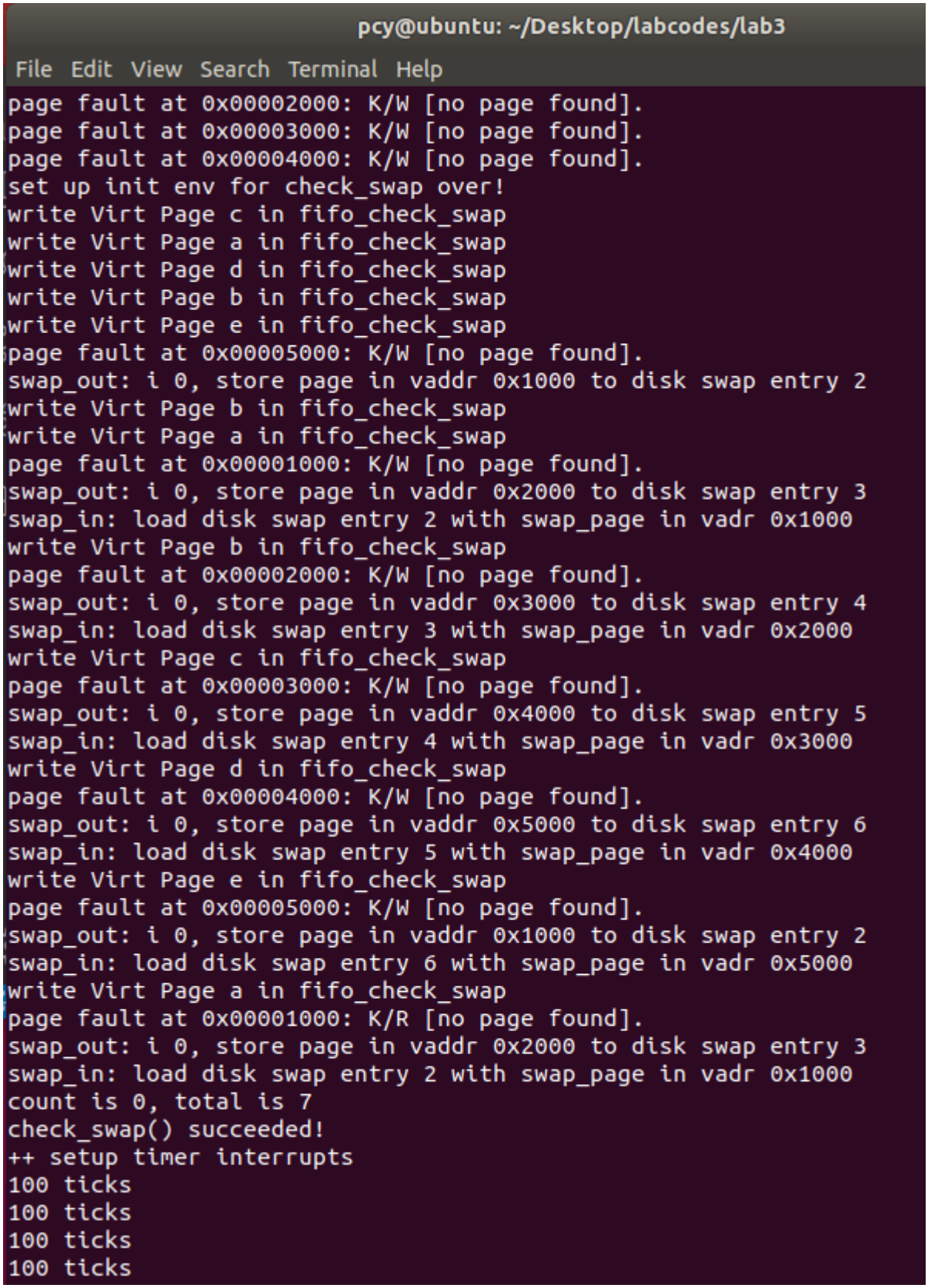
回答如下问题：

- 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？——能够支持
- 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？

    首选 页表项的 Dirty Bit 为 0 的页 且 Access Bit 为 0 的页，其次是 访问了但没修改的页，最次是 访问了修改了的页。
  - 在ucore中如何判断具有这样特征的页？

    !(*ptep & PTE_A) && !*(ptep & PTE_D) 没被访问过 也没被修改过 (*ptep & PTE_A) && !*(ptep & PTE_D) 被访问过 但没被修改过 !(*ptep & PTE_A) && *(ptep & PTE_D) 没被访问过 但被修改过
  - 何时进行换入和换出操作？

    换入是在缺页异常的时候，换出是在物理页帧满的时候。


pcy@ubuntu: ~/Desktop/labcodes/lab3
File Edit View Search Terminal Help
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
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
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

```
pcy@ubuntu:~/Desktop/labcodes/lab3$ make grade
Check SWAP:                (5.4s)
  -check pmm:                                  OK
  -check page table:                           OK
  -check vmm:                                  OK
  -check swap page fault:                      OK
  -check ticks:                                OK
Total Score: 45/45
```

“虚拟”的作用或意义：

1. 通过设置页表项来限定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。
2. 在实际访问某虚拟内存地址时，操作系统动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，按需分页（demand paging）
3. 把不经常访问的数据所占的内存空间临时写到硬盘上，当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，页换入换出（page　swap in/out）。更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

在ucore中描述应用程序对虚拟内存“需求”的数据结构是vma_struct，vma_struct中：vm_start和vm_end描述了一个连续地址的虚拟内存空间的起始位置和结束位置，这两个值都应该是PGSIZE 对齐的，而且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，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，这里把一个mm_struct结构的变量简称为mm变量。这个数据结构表示了包含所有虚拟内存空间的共同属性，mm_struct中：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。

一个扇区的大小为512（2\^8）字节，所以需要8个连续扇区才能放置一个4KB的页。

扩展练习 Challenge 1：实现识别dirty bit的 extended clock页替换算法

改进的时钟算法的主要思想是：

1. 为每个页设置两个标志位，一个是accessed位表示访问位，一个是dirty位表示被写位。
2. 维护一个与物理页面数目相等的环形链表，发生访问时，如果要访问的页存在，则将accessed位置为1，如果访问的同时也进行了修改，则将dirty位变为1。
3. 如果发生了缺页，则从指针的位置开始循环寻找，同时对accessed位和dirty位进行修改，修改的方式是：

```
    accessed  dirty
    (1,1)- > (0,1)
    (1,0)- > (0, 0)
    (0,1)- > (0,0)
    (0,0) 置换
```

当找到（0，0）时置换此页，并将指针下移。

设计实现说明：

对页的访问和修改信息可以在二级页表中的pte得到，同时在本算法中需要对标志位进行修改，也是对与之对应的页表项进行修改。 两个标志位在原本的ucore代码里已经实现了，在mmu.h文件中：

```
#define PTE_A           0x020                   // Accessed
#define PTE_D           0x040                   // Dirty
```

对照swap_fifo.h和swap_fifo.c，编写swap_extended_clock.h和swap_extended_clock.c。并swap.c中加入头文件swap_extended_clock.h，在swap_init(void)中将sm改成swap_manager_extended_clock。

swap_extended_clock.c：

```
#include <defs.h>
#include <x86.h>
#include <stdio.h>
#include <string.h>
#include <swap.h>
#include <swap_extended_clock.h>
#include <list.h>

//维护一个与物理页面数目相等的环形链表，发生访问时，
//如果要访问的页存在，则将accessed位置为1，如果访问的同时也进行了修改，则将dirty位变为1。
//维护的环形链表为`pra_list_head`，头指针为`clock_p`
list_entry_t *clock_p;
list_entry_t pra_list_head;


//当访问页时，相应的A位和D位都会被CPU修改
//不用操作系统进行专门修改，所以只要对发生缺页时的情况进行处理即可。

static int
_extended_clock_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     // 将头指针指向pra_list_head的起始地址
     clock_p = (list_entry_t*)&pra_list_head;
     //cprintf(" mm->sm_priv %x in extended_clock_init_mm\n",mm->sm_priv);
     return 0;
}
/*
 * (3)_extended_clock_map_swappable: According extended_clock PRA, we should link the
most recent arrival page at the back of pra_list_head qeueue
 */
static int
_extended_clock_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);
    //将换进的页加入链表
    list_add_before(clock_p, entry);
    //pte_t *pte = get_pte(mm->pgdir, page2kva(page), 0);
    pte_t *pte = get_pte(mm->pgdir, addr, 0);
```

```
    //int access = (*pte)&(PTE_A)?1:0;
    //int dirty = (*pte)&(PTE_D)?1:0;
    return 0;
}
/*
如果发生了缺页，则从指针的位置开始循环寻找，同时对accessed位和dirty位进行修改，修改的方式是：
    accessed  dirty
    (1,1)- > (0,1)
    (1,0)- > (0, 0)
    (0,1)- > (0,0)
    (0,0) 置换
当找到（0，0）时置换此页，并将指针下移。
*/
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 = clock_p;
     le = head->next;
     cprintf("\n---start---\n");
     while (1) {
         struct Page *page = le2page(le, pra_page_link);
         pte_t * pte = get_pte(mm->pgdir, page->pra_vaddr, 0);
         int accessed = (*pte)&(PTE_A)?1:0;
         int dirty = (*pte)&(PTE_D)?1:0;
         if (le==clock_p)
             cprintf("->");
         else
             cprintf("  ");
         cprintf("clock state: 0x%4x: A:%x, D:%x\n", page->pra_vaddr, accessed, dirty);
         le = le->next;
         if (le == head) {
             break;
         }
     }
     cprintf("----end----\n");

     le = clock_p;
     while (1) {
         if (le == head) {
             le = le->next;
             clock_p = clock_p -> next;
         }
         struct Page *page = le2page(le, pra_page_link);
         pte_t * pte = get_pte(mm->pgdir, page->pra_vaddr, 0);
         int accessed = (*pte)&(PTE_A)?1:0;
         int dirty = (*pte)&(PTE_D)?1:0;
         if (accessed) {//如果A为1，则将其变为0;
             cprintf("clock state: 0x%4x: A:%x, D:%x\n", page->pra_vaddr, accessed,
dirty);
```

```
            (*pte) = (*pte) & (~PTE_A);
            cprintf("\tclock state: 0x%4x: A:%x, D:%x\n",page->pra_vaddr, (*pte)&
(PTE_A)?1:0, (*pte)&(PTE_D)?1:0);
        }
        else if (!accessed && dirty) {//如果A为0，D为1 ，则D变为0;
            cprintf("clock state: 0x%4x: A:%x, D:%x\n", page->pra_vaddr, accessed,
dirty);
            (*pte) = (*pte) & (~PTE_D);
            cprintf("\tclock state: 0x%4x: A:%x, D:%x\n", page->pra_vaddr, (*pte)&
(PTE_A)?1:0, (*pte)&(PTE_D)?1:0);
        } else if (!accessed && !dirty){//如果都为0，则置换此页，且clock_p指针下移一位;
            struct Page *p = le2page(le, pra_page_link);
            list_del(le);
            clock_p = clock_p->next;
            assert(p !=NULL);
            *ptr_page = p;

            le = head->next;
            cprintf("\n--after--start---\n");
            while (1) {
                struct Page *page = le2page(le, pra_page_link);
                pte_t * pte = get_pte(mm->pgdir, page->pra_vaddr, 0);
                int accessed = (*pte)&(PTE_A)?1:0;
                int dirty = (*pte)&(PTE_D)?1:0;
                if (le==clock_p)
                    cprintf("->");
                else
                    cprintf("  ");
                cprintf("clock state: 0x%4x: A:%x, D:%x\n", page->pra_vaddr, accessed,
dirty);
                le = le->next;
                if (le == head) {
                    break;
                }
            }
            cprintf("--after--end----\n");
            return 0;
        }
        le = le->next;
        clock_p = clock_p->next;
    }

}
static int
_extended_clock_check_swap(void) {
    //初始状态置入了abcd四页，然后进行了写e，读c，写d，读a，写b，
    //写e，写c，写e，读c，写e，写a，写a，读b，读c，读d，写e，读a，写b，写e 这些操作
    unsigned char tmp;
    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x1e;

    cprintf("read Virt Page c in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x3000;
```

```
    //cprintf("tmp = 0x%4x\n", tmp);

    cprintf("write Virt Page d in extended_clock_check_swap\n");
    *(unsigned char *)0x4000 = 0x0a;

    cprintf("read Virt Page a in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x1000;
    //cprintf("tmp = 0x%4x\n", tmp);


    cprintf("write Virt Page b in extended_clock_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;

    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x1e;

    cprintf("write Virt Page c in extended_clock_check_swap\n");
    *(unsigned char *)0x3000 = 0x0e;

    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x2e;

    cprintf("read Virt Page c in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x3000;

    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x2e;
    //cprintf("--------\n");
    cprintf("write Virt Page a in extended_clock_check_swap\n");
    *(unsigned char *)0x1000 = 0x1a;
    cprintf("write Virt Page a in extended_clock_check_swap\n");
    *(unsigned char *)0x1000 = 0x1a;

    //cprintf("--------\n");
    cprintf("read Virt Page b in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x2000;
    //cprintf("tmp = 0x%4x\n", tmp);
    //cprintf("--------\n");

    cprintf("read Virt Page c in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x3000;
    //cprintf("tmp = 0x%4x\n", tmp);

    cprintf("read Virt Page d in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x4000;
    //cprintf("tmp = 0x%4x\n", tmp);

    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;

    cprintf("read Virt Page a in extended_clock_check_swap\n");
    tmp = *(unsigned char *)0x1000;
    //cprintf("tmp = 0x%4x\n", tmp);
```

```
    cprintf("write Virt Page b in extended_clock_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;

    cprintf("write Virt Page e in extended_clock_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;


    return 0;
}
static int
_extended_clock_init(void)
{
    return 0;
}
static int
_extended_clock_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}
static int
_extended_clock_tick_event(struct mm_struct *mm)
{ return 0; }
struct swap_manager swap_manager_extended_clock =
{
     .name            = "extended_clock swap manager",
     .init            = &_extended_clock_init,
     .init_mm         = &_extended_clock_init_mm,
     .tick_event      = &_extended_clock_tick_event,
     .map_swappable   = &_extended_clock_map_swappable,
     .set_unswappable = &_extended_clock_set_unswappable,
     .swap_out_victim = &_extended_clock_swap_out_victim,
     .check_swap      = &_extended_clock_check_swap,
};
```

测试替换算法时，控制台的输出信息：

```
SWAP: manager = extended_clock swap manager
BEGIN check_swap: count 1, total 31963
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 e in extended_clock_check_swap
page fault at 0x00005000: K/W [no page found].

---start---
  clock state: 0x1000: A:1, D:1
  clock state: 0x2000: A:1, D:1
```

```
  clock state: 0x3000: A:1, D:1
  clock state: 0x4000: A:1, D:1
----end----
clock state: 0x1000: A:1, D:1
    clock state: 0x1000: A:0, D:1
clock state: 0x2000: A:1, D:1
    clock state: 0x2000: A:0, D:1
clock state: 0x3000: A:1, D:1
    clock state: 0x3000: A:0, D:1
clock state: 0x4000: A:1, D:1
    clock state: 0x4000: A:0, D:1
clock state: 0x1000: A:0, D:1
    clock state: 0x1000: A:0, D:0
clock state: 0x2000: A:0, D:1
    clock state: 0x2000: A:0, D:0
clock state: 0x3000: A:0, D:1
    clock state: 0x3000: A:0, D:0
clock state: 0x4000: A:0, D:1
    clock state: 0x4000: A:0, D:0

--after--start---
->clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
  clock state: 0x4000: A:0, D:0
--after--end----
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
read Virt Page c in extended_clock_check_swap
write Virt Page d in extended_clock_check_swap
read Virt Page a in extended_clock_check_swap
page fault at 0x00001000: K/R [no page found].

---start---
  clock state: 0x5000: A:1, D:1
->clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
  clock state: 0x4000: A:0, D:0
----end----

--after--start---
  clock state: 0x5000: A:1, D:1
->clock state: 0x3000: A:0, D:0
  clock state: 0x4000: A:0, D:0
--after--end----
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 extended_clock_check_swap
page fault at 0x00002000: K/W [no page found].

---start---
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
->clock state: 0x3000: A:0, D:0
  clock state: 0x4000: A:0, D:0
```

```
----end----

--after--start---
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
->clock state: 0x4000: A:0, D:0
--after--end----
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 e in extended_clock_check_swap
write Virt Page c in extended_clock_check_swap
page fault at 0x00003000: K/W [no page found].

---start---
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
  clock state: 0x2000: A:1, D:1
->clock state: 0x4000: A:0, D:0
----end----

--after--start---
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
  clock state: 0x2000: A:1, D:1
--after--end----
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 e in extended_clock_check_swap
read Virt Page c in extended_clock_check_swap
write Virt Page e in extended_clock_check_swap
write Virt Page a in extended_clock_check_swap
write Virt Page a in extended_clock_check_swap
read Virt Page b in extended_clock_check_swap
read Virt Page c in extended_clock_check_swap
read Virt Page d in extended_clock_check_swap
page fault at 0x00004000: K/R [no page found].

---start---
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:1
  clock state: 0x2000: A:1, D:1
  clock state: 0x3000: A:1, D:1
----end----
clock state: 0x5000: A:1, D:1
    clock state: 0x5000: A:0, D:1
clock state: 0x1000: A:1, D:1
    clock state: 0x1000: A:0, D:1
clock state: 0x2000: A:1, D:1
    clock state: 0x2000: A:0, D:1
clock state: 0x3000: A:1, D:1
    clock state: 0x3000: A:0, D:1
clock state: 0x5000: A:0, D:1
    clock state: 0x5000: A:0, D:0
```

```
clock state: 0x1000: A:0, D:1
    clock state: 0x1000: A:0, D:0
clock state: 0x2000: A:0, D:1
    clock state: 0x2000: A:0, D:0
clock state: 0x3000: A:0, D:1
    clock state: 0x3000: A:0, D:0

--after--start---
->clock state: 0x1000: A:0, D:0
  clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
--after--end----
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 extended_clock_check_swap
page fault at 0x00005000: K/W [no page found].

---start---
  clock state: 0x4000: A:1, D:0
->clock state: 0x1000: A:0, D:0
  clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
----end----

--after--start---
  clock state: 0x4000: A:1, D:0
->clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
--after--end----
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
read Virt Page a in extended_clock_check_swap
page fault at 0x00001000: K/R [no page found].

---start---
  clock state: 0x4000: A:1, D:0
  clock state: 0x5000: A:1, D:1
->clock state: 0x2000: A:0, D:0
  clock state: 0x3000: A:0, D:0
----end----

--after--start---
  clock state: 0x4000: A:1, D:0
  clock state: 0x5000: A:1, D:1
->clock state: 0x3000: A:0, D:0
--after--end----
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 extended_clock_check_swap
page fault at 0x00002000: K/W [no page found].

---start---
  clock state: 0x4000: A:1, D:0
```

```
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
->clock state: 0x3000: A:0, D:0
----end----

--after--start---
  clock state: 0x4000: A:1, D:0
  clock state: 0x5000: A:1, D:1
  clock state: 0x1000: A:1, D:0
--after--end----
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 e in extended_clock_check_swap
count is 0, total is 7
check_swap() succeeded!
```