《操作系统原理》lab4实验报告

专业：计算机科学与技术

姓名：郝裕玮

学号：18329015

1. 实验目的

①了解虚拟内存的 Page Fault 异常处理实现

②了解页替换算法在操作系统中的实现

2. 实验过程与结果

练习 0：填写已有实验

本实验依赖实验 1 和 3。请把你做的实验 1 和 3 的代码填入本实验中代码中有“LAB1”,“LAB2”的注释相应部分。

答：需要修改的有 debug/kdebug.c，trap/trap.c，mm/default_pmm.c，mm/pmm.c 这四个文件，使用 meld 进行修改即可。

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？

答：本题对于 do_pgfault 函数的补充内容要求为：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。

所以本题的代码设计思路为：

①检查页表（PT）中是否有相应的页表项（PTE），若页表项为空，则说明没有尚未生成映射，此时我们需要创建一个含有该 PTE 的 PT。

②若物理地址 pa 不存在，则使用 pgdir_alloc_page 获取一个物理页，并将逻辑地址和物理地址作映射。

所以本题需补充的代码如下所示（对代码的分析已全部包含在注释中）：

```
    ptep=get_pte(mm->pgdir,addr,1);//获取 ptep
    //get_pte:获得一个 pte 并返回这个 pte 的内核虚拟地址，如果这个 pte 不存在，则为 PT 分配一个页面
    //(1) try to find a pte, if pte's PT(Page Table) isn't existed, then create a PT.
    //所以对于(1)的英文注释，只需调用 get_pte 函数即可
    if(ptep==NULL){
        goto failed;//若 pte 不存在且分配页面失败则跳转至 failed 部分返回 ret
    }
    if(*ptep==0){//如果是上述新创建的二级页表，那么*ptep 就为 0，代表页表为空。
    //此时需调用 pgdir_alloc_page，对它进行初始化
    //若 PTE 所指向的物理页表地址不存在，则分配一个物理页并将逻辑地址和物理地址作映射(即让 PTE 指向物理页帧)
        if(pgdir_alloc_page(mm->pgdir,addr,perm)==NULL){
            //调用 alloc_page 和 page_insert 函数来分配一个页面大小的内存，并用线性地址 addr 和 mm->pgdir 来设置一个映射关系 mm->pgdir<--->addr
            //perm 设置物理页权限，保证与其对应的虚拟页的权限一致
            //分配物理页，并与对应的虚拟页建立映射关系
            //(2) if the phy addr isn't exist, then alloc a page & map the phy addr with logical addr
            //所以对于(2)的英文注释，调用 pgdir_alloc_page 函数即可
            goto failed;//分配或映射失败则跳转至 failed 部分返回 ret
        }
    }
```

执行 make qemu 后，结果如下：

```
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0108b10 (phys)
  edata  0xc011fa68 (phys)
  end    0xc0120bb0 (phys)
Kernel executable memory footprint: 131KB
ebp:0xc011ef38 eip:0xc01009df args:0x00010094 0x00000000 0xc011ef68 0xc01000cb
    kern/debug/kdebug.c:297: print_stackframe+21
ebp:0xc011ef48 eip:0xc0100cce args:0x00000000 0x00000000 0x00000000 0xc011efb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc011ef68 eip:0xc01000cb args:0x00000000 0xc011ef90 0xffff0000 0xc011ef94
    kern/init/init.c:57: grade_backtrace2+33
ebp:0xc011ef88 eip:0xc01000f4 args:0x00000000 0xffff0000 0xc011efb4 0x0000002a
    kern/init/init.c:62: grade_backtrace1+38
ebp:0xc011efa8 eip:0xc0100112 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:67: grade_backtrace0+23
ebp:0xc011efc8 eip:0xc0100137 args:0xc0108b3c 0xc0108b20 0x00001148 0x00000000
    kern/init/init.c:72: grade_backtrace+34
ebp:0xc011eff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:32: kern_init+84
memory management: default_pmm_manager
```

```
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- 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 31996, total 31996
```

```
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
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
page fault at 0x000000ae: K/R [no page found].
not valid addr ae, and  can not find it in vma
trapframe at 0xc011ecf4
  edi  0x00000001
  esi  0x00000000
  ebp  0xc011ed70
  oesp 0xc011ed14
  ebx  0x00007cfc
  edx  0xc0302000
  ecx  0x00005000
  eax  0x00000092
```

```
  ds   0x----0010
  es   0x----0010
  fs   0x----0023
  gs   0x----0023
  trap 0x0000000e Page Fault
  err  0x00000000
  eip  0xc010620f
  cs   0x----0008
  flag 0x00000046 PF,ZF,IOPL=0
kernel panic at kern/trap/trap.c:182:
    handle pgfault failed. invalid parameter 代码未完全补齐，所以报错

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> [~/Desktop/lab4]
```

由上图的红色方框部分可知，出现了“check_pgfault() succeeded!”，表明通过了 check_pgfault 函数的测试，所以练习 1 补充代码正确。

请回答如下问题：

（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。

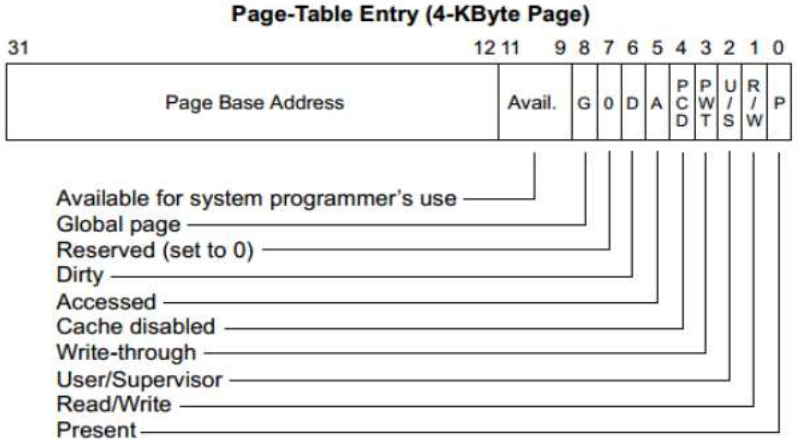
答：查看 mmu.h 可知页目录项和页表项的组成部分：

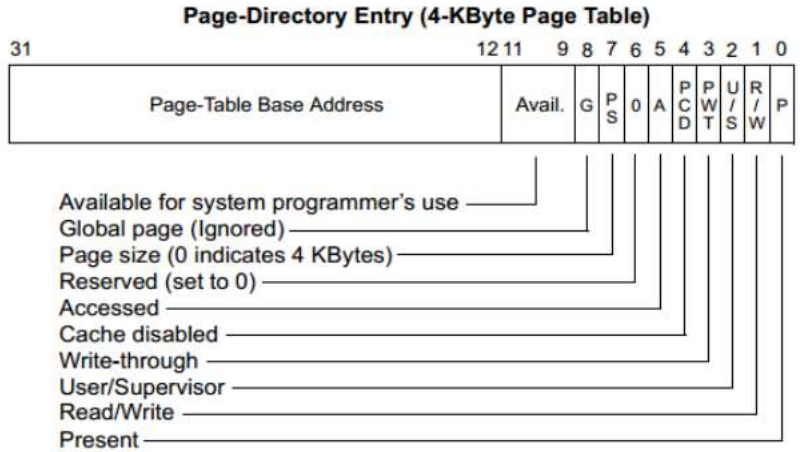
```
/* page table/directory entry flags */
#define PTE_P           0x001                   // Present
#define PTE_W           0x002                   // Writeable
#define PTE_U           0x004                   // User
#define PTE_PWT         0x008                   // Write-Through
#define PTE_PCD         0x010                   // Cache-Disable
#define PTE_A           0x020                   // Accessed
#define PTE_D           0x040                   // Dirty
#define PTE_PS          0x080                   // Page Size
#define PTE_MBZ         0x180                   // Bits must be zero
#define PTE_AVAIL       0xE00                   // Available for softwa
re use
                                                // The PTE_AVAIL bits a
ren't used by the kernel or interpreted by the
                                                // hardware, so user pr
ocesses are allowed to set them arbitrarily.

#define PTE_USER        (PTE_U | PTE_W | PTE_P)
```

所以可总结出以下表格：

名称	含义
PTE_P	存在位
PTE_w	可写控制位
PTE_U	该页访问需要的特权级（用户或内核）
PTE_PWT	直写（Write-through）控制位
PTE_PCD	缓存控制位
PTE_A	访存控制位
PTE_D	脏位
PTE_PS	页大小
PTE_MBZ	必须为 0
PTE_AVAIL	设置内核或系统中断

同时，页目录项和页表项的格式如下所示：





对 ucore 实现页替换算法的潜在用处：

①PDE(页目录项)的高二十位表示该 PDE 对应的页表起始位置，即物理地址。

②PTE(页表项)的高二十位表示该 PTE 条目指向的物理页的物理地址。

③页目录项的作用：一级页表，存储了各二级页表的起始地址。页表是二级页表，存储了各个页的起始地址，即用于索引页表。

④页表项的作用：二级页表，存储各个页的起始地址。通过页目录项和页表项就可以将一个虚拟地址（线性地址）翻译为物理地址，即用于记录虚拟页在磁盘中的位置。

⑤对于页替换来说，它主要的操作就是换入和换出。而页目录项和页表项为这两种操作提供了磁盘的位置信息（页表项提供虚拟页在磁盘中的位置，页目录项用于索引页表）。且上述的这些页目录项和页表项的功能位（具体功能可见表格）对于分析内存情况和记录内存信息有重要作用。

（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？

答：①调用中断机制，引起中断；

②将引发页访问异常的线性地址保存在 cr2 寄存器中；

③设置 error_code（错误代码），将其压入中断栈中；

④因为 ISR（中断服务程序）是在内核态下运行的，所以无需进行上下文切换。只需要根据中断源，跳转到缺页服务例程，调用 do_pgfault 函数进行处理即可。

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" 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？

答：①对于需要进一步完善的 do_pgfault 函数，可见 LAB 3 EXERCISE 2：

YOUR CODE 部分，中文注释如下所示：

```
    else {
    /*LAB3 EXERCISE 2: YOUR CODE
    * 现在我们认为这个 pte 是一个交换条目，我们应该将数据从磁盘加载到一个带有 phy addr 的页面，
    * 并将 phy addr 与 logical addr 映射，触发交换管理器来记录该页面的访问情况。
    *
    *  一些有用的宏和定义，您可以在下面的实现中使用它们。
    *  宏或函数:
    *    swap_in(mm, addr, &page) : 分配一个内存页，然后根据 PTE 中 addr 的交换项，找到磁盘页的 addr，将磁盘页的内容读入该内存页
    *    page_insert :  用线性地址 la 建立一个 Page 的 phy addr 的映射
    *    swap_map_swappable :  设置页面可切换
    */
        if(swap_init_ok) {
            struct Page *page=NULL;
                                    //(1）根据 mm 和 addr，尝试将右侧磁盘页的内容加载到该页所管理的内存中。
```

```
                                        //(2) 根据 mm,addr 和 Page，设置物理地
址<--->逻辑地址的映射
                                        //(3) 设置页面可切换
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
    }
```

根据上述注释及其函数，宏和定义的用法解析，即可补全完整代码（对代码的分析已全部包含在注释中）：

```
    else{//若*ptep!=0,则代表 pa 不为空，即页表项不为空，于是准备向内存中换入
该页
        if(swap_init_ok){//代表初始化成功
            struct Page* page=NULL;
            ret=swap_in(mm,addr,&page);//根据 mm 和 addr 将磁盘中的内容读入
到该内存页 page 中
            //(1)According to the mm AND addr, try to load the content
of right disk page into the memory which page managed.
            if(ret!=0){
                goto failed;//若换页失败则跳转至 failed 部分并返回 ret
            }
            page_insert(mm->pgdir,page,addr,perm);//用线性地址 addr 建立一
个 Page 的物理地址和虚拟地址之间的映射，并用 perm 设置物理页权限
            //(2) According to the mm, addr AND page, setup the map of
phy addr <---> logical addr
            swap_map_swappable(mm,addr,page,1); //将该页设置为可交换
            //(3) make the page swappable.
            page->pra_vaddr=addr;//设置页对应的虚拟地址
        }
        else{//若初始化失败
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;//跳转至 failed 部分并返回 ret
        }
    }
```

②swap_fifo.c 中的 map_swappable 函数：

根据注释即可补全该函数（对代码的分析已全部包含在注释中）：

```
/*
 * (3)_fifo_map_swappable:根据 FIFO PRA，我们应该在 pra_list_head 队列的后面链接最近到达的页面
 */
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);//对应英文注释
(1)link the most recent arrival page at the back of the pra_list_head queue.
    //将最近到达的节点 head 加入到队列当中
    //节点加入位置为头节点和上一个加入的节点的中间位置
    //所以 entry->next 永远是最新进入的节点，又因为是双向链表，所以 entry->prev(即链表尾部)永远是最早进入的节点
    return 0;
}
```

③swap_fifo.c 中的 swap_out_victim 函数：

根据注释即可补全该函数（对代码的分析已全部包含在注释中）：

```
/*
 *  (4)_fifo_swap_out_victim: 根据 FIFO PRA，我们应该断开 pra_list_head 队列前面最早到达的页面的链接，然后将该页 addr 中的 addr 设置为 ptr_page。
 */
static int
_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
{
    //对于 FIFO 算法，每次换出的都是最早进入的页(对于链表来说就是最早进入的节点)
    list_entry_t *head=(list_entry_t*) mm->sm_priv;//找到链表头
    assert(head!=NULL);//判断链表是否为空，为空则终止程序
    assert(in_tick==0);//补充前原代码里就有的，暂未查明具体用途
```

```
    /* Select the victim */
    /*LAB3 EXERCISE 2: YOUR CODE*/
    list_entry_t *prev_page=head->prev;//找到最早进入队列的节点
    //因为是双向链表，所以 head->prev(即链表尾部)永远是最早进入的节点
    assert(head!=prev_page);//判断是否就是头节点对应的页
    struct Page *page=le2page(prev_page,pra_page_link);//得到节点所属的
Page 结构
    list_del(prev_page);//对应英文注释
(1)  unlink the  earliest arrival page in front of pra_list_head queue
    //从链表上删除刚刚找到的最早进入队列的节点 prev_page
    assert(page!=NULL);//判断 page 是否为空，为空则终止程序
    *ptr_page=page;//对应英文注释
(2) set the addr of addr of this page to ptr_page
    //将这一页的地址存储到 ptr_page 中
    return 0;
}
```

执行 make qemu 后，结果如下：

```
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0108c0c (phys)
  edata  0xc0120a68 (phys)
  end    0xc0121bb0 (phys)
Kernel executable memory footprint: 135KB
ebp:0xc011ff38 eip:0xc01009df args:0x00010094 0x00000000 0xc011ff68 0xc01000cb
    kern/debug/kdebug.c:297: print_stackframe+21
ebp:0xc011ff48 eip:0xc0100cce args:0x00000000 0x00000000 0x00000000 0xc011ffb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc011ff68 eip:0xc01000cb args:0x00000000 0xc011ff90 0xffff0000 0xc011ff94
    kern/init/init.c:57: grade_backtrace2+33
ebp:0xc011ff88 eip:0xc01000f4 args:0x00000000 0xffff0000 0xc011ffb4 0x0000002a
    kern/init/init.c:62: grade_backtrace1+38
ebp:0xc011ffa8 eip:0xc0100112 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:67: grade_backtrace0+23
ebp:0xc011ffc8 eip:0xc0100137 args:0xc0108c3c 0xc0108c20 0x00001148 0x00000000
    kern/init/init.c:72: grade_backtrace+34
ebp:0xc011fff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:32: kern_init+84
memory management: default_pmm_manager
```

```
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- 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 31995, total 31995
```

```
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
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!
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:20:
    EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

由上图的红色方框部分可知，出现了“check_swap() succeeded!”，表明通过了 check_swap 函数的测试，所以练习 2 补充代码正确。

请在实验报告中回答如下问题：

（1）如果要在 ucore 上实现" 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。

答：可以，以下是具体步骤：

①先将页用环形链表连接起来（即首尾相连）；

②设置当前指针指向最先进入的页；

③设置每个页表项的 PTE_A 和 PTE_D（即访问位和脏位，判断该页是否被访问或修改过），若该页被访问过，PTE_A 则为 1，反之为 0（因为一开始所有页面均未被访问过，所以均初始化为 0）。若该页被修改过，PTE_D 则为 1，反之

为 0（因为一开始所有页面均未被访问过，所以均初始化为 0）。

④从最先进入的页开始对环形链表进行遍历扫描，我们可根据 PTE_A 和 PTE_D 的 4 种取值组合得出不同的操作，具体如下述表格所示：

PTE_A	PTE_D	操作
0	0	将该页从链表中删除，随后继续扫描遍历（因为未修改，所以无需写入外存）
0	1	将该页的 PTE_D 修改为 0，并将该页写入到外存中，随后继续扫描遍历
1	0	将该页的 PTE_A 修改为 0，随后继续扫描遍历
1	1	将该页的 PTE_A 和 PTE_D 均修改为 0，并将该页写入到外存中，随后继续扫描遍历

（2）需要被换出的页的特征是什么？

答: ①该页尚未被访问过（或者该页脏位 PTE_D 在上一次扫描遍历环形链表时从 1 被修改为 0）；

②该页内容尚未被修改过（即与外存中的对应数据保持一致）。

（3）在 ucore 中如何判断具有这样特征的页？

答：PTE_A 和 PTE_D 均为 0 的页（即未被访问过也未被修改过）。

（4）何时进行换入和换出操作？

答：需要的页不在页表中且页表已满时，需要进行换入换出操作。

3. 实验感想

本次实验较为简单，代码量较小，思考难度也较低，所以感想相较于前三次实验可能会少一些。

本次实验最重要的部分就是需要理解 FIFO 替换算法以及算法的实现方式。在此次代码中，实现方式是采用双向循环链表的方式来实现（使用 list_entry_t 这一数据结构）。

同时我认为本次实验的重点是 swap_fifo.c 中的 map_swappable 函数的补充。虽然只补充了一行代码：

```
    list_add(head,entry);
```

但实际上这个函数正是实现 FIFO 的关键所在，因为在这里，list_add 函数是将 head 节点插入到头节点 entry 和上一个刚加入节点之间的位置。也就是保证头节点的下一个节点（entry->next）永远是最新加入的节点，而头节点的上一个节点（即 entry->prev 是链表的尾结点）永远是最先加入的节点。这样的链表结构就便于我们后续的换入换出操作。

本次实验也有一个小问题暂未解决，就是 swap_out_victim 函数中的这一行意思尚未明白：

```
assert(in_tick==0);//补充前原代码里就有的，暂未查明具体用途
```

在网上查阅资料和查看源码均为找到对其的详细解释，希望能在以后的学习中解决这一问题。同时因为这行代码是之前就有的，所以并不影响我对于代码的后续补充和运行结果。

通过本次实验，我了解并掌握了物理内存管理中的连续空间分配算法 FIFO 的具体实现以及如何建立二级页表。又因为本次实验是建立在我们的上次实验 lab3 的基础之上，所以将两次实验结合起来学习，使我对虚拟内存和物理内存之间的关系了解的更加透彻，明白了内存之间相互交换，定位，读入读出的原理和机制。明白了虚拟内存对于我们的重要性，它使得我们可以为操作系统提供比实际内存大的多的内存空间。