Lab 3

于成俊、楚乾靖、李帅东

一、题目

练习1：理解基于FIFO的页面替换算法（思考题）

描述FIFO页面置换算法下，一个页面从被换入到被换出的过程中，会经过代码里哪些函数/宏的处理（或者说，需要调用哪些函数/宏），并用简单的一两句话描述每个函数在过程中做了什么？（为了方便同学们完成练习，所以实际上我们的项目代码和实验指导的还是略有不同，例如我们将FIFO页面置换算法头文件的大部分代码放在了`kern/mm/swap_fifo.c`文件中，这点请同学们注意）

- 至少正确指出10个不同的函数分别做了什么？如果少于10个将酌情给分。我们认为只要函数原型不同，就算两个不同的函数。要求指出对执行过程有实际影响,删去后会导致输出结果不同的函数（例如assert）而不是cprintf这样的函数。如果你选择的函数不能完整地体现”从换入到换出“的过程，比如10个函数都是页面换入的时候调用的，或者解释功能的时候只解释了这10个函数在页面换入时的功能，那么也会扣除一定的分数
- 首先说明虚拟内存管理初始化的相关文件/函数和对应功能。

· **vmm_init**：建立虚拟内存到物理内存的映射关系

ide_init：初始化swap硬盘，在内存中把数据来回复制

swap_init：初始化页面置换算法

完成以上初始化后，就可以在发生缺页异常时调用（这个调用在trap.c文件中执行）相应缺页异常处理函数以处理缺页异常了

- **触发缺页异常(trap.c/pgfault_handler())**：触发缺页异常时，内核会调用该函数去处理，该函数会调用相关函数，其另外做的事情只是输出缺页提示信息。
- **处理缺页异常(vmm.c/do_pgfault())**：定义了缺页异常处理的主要逻辑行为，依次包括如下行为。
  - **检查虚拟地址合法性(vmm.c/find_vma())**：检查请求虚拟地址是否满足实际虚拟地址范围，然后查找该虚拟地址是否真实存在
  - **找到/创建页表项(pmm.c/get_pte())**：根据虚拟地址和页目录表来获取对应的页表项。当一级页表和二级页表不存在时需要创建它们。
  - `swap.c\swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)`被换入：分配一个空闲的物理页面，再根据虚拟地址获取到对应的页表项，在页表项的信息，从磁盘中读取数据到物理页面当中
    - `alloc_page()`分配页面，当做内存页，用来存储从磁盘上读取的数据
    - `assert(result!=NULL)`确保要分配到页面，否则后续没法将磁盘上的数据转移到内存
    - `get_pte(mm->pgdir,addr,0)`根据传入的la在不创建页表项的情况下，找到la所对应的页表项，并返回其内核虚拟地址
    - `swapfs_read((*ptep),result)!=0`将数据从磁盘读取到内存，并确保读取正确
    - **必要时换出页面(swap.c/swap_out())**：首先需要确定需要换出的内存页，然后写回数据到磁盘，最后释放该内存页。注意要刷新快表。

`_fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)`

确保不在时钟中断处理中调用该函数时，获取FIFO队列中最早进入的页面，删除。

- `assert(head!=NULL)` 确保head=(**list_entry_t***) mm->sm_priv,因此要确保是存在链表头的
- `assert(in_tick==0)` in_tick =0时，代表不是在时钟中断处理函数中调用的该函数
- `list_entry_t* entry = list_prev(head)` 双向链表中，获取头结点的前一个节点
- `list_del(entry)` 对于选区中的页面，从链表中删除!

`swap.c\swap_out(struct mm_struct *mm, int n, int in_tick)`

对于要换出n页，执行n轮次如下操作：利用`ss->swap_out_victim`寻找受害页，如果找到受害页，根据其虚拟地址，得到对应的页表项。将物理页上的内容以扇区为单位填写至磁盘当中，最后更新TLB

- `swap_out_victim(mm,&page,in_tick)` 在本题目当中，是利用FIFO算法寻找换页牺牲者
- `get_pte(mm->pgdir,v,0)` 获取到牺牲页v的内核虚拟地址
- `assert((*ptep & PTE_V)!=0)` 确保这个页是有效的
- `swapfs_write((page->pra_vaddr/PGSIZE+1)<<8, page)` 以扇区为单位将牺牲页上的数据写到磁盘上的对应位置。
- `tlb_invalidate(mm->pgdir,v)` 刷新TLB快表，使TLB有效

练习2：深入理解不同分页模式的工作原理（思考题）

get_pte()函数（位于`kern/mm/pmm.c`）用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作。这在操作系统中的分页机制下，是实现虚拟内存与物理内存之间映射关系非常重要的内容。

- get_pte()函数中有两段形式类似的代码， 结合sv32，sv39，sv48的异同，解释这两段代码为什么如此相像。
- 目前get_pte()函数将`页表项的查找`和`页表项的分配`合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？

get_pte中有两段形式类似的代码：

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /*获取到  la对应的  页表项，返回内核虚拟地址*/
    pde_t *pdep1 = &pgdir[PDX1(la)];//找到页目录项pdep1
    if (!(*pdep1 & PTE_V)) {
        struct Page *page;
        //如果不合法，就要分配物理页
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        set_page_ref(page, 1);//物理页有一个对应的虚拟页
        uintptr_t pa = page2pa(page);//获取到页的物理地址
        memset(KADDR(pa), 0, PGSIZE);//将对应的内核虚拟地址赋值为0
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//创建页表项，传入物理页号和标志位UV
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//找到2级页表项对应的页表
//    pde_t *pdep0 = &((pde_t *)(PDE_ADDR(*pdep1)))[PDX0(la)];
    if (!(*pdep0 & PTE_V)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
```

```
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
 //     memset(pa, 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

两段代码分别对应一级页表、二级页表，由于它们只是等级上的不同、而结构完全相同，所以代码相似。都包括：获取对应页、不存在则创建、设置页属性及其内存状态、构造页表项。

我认为挺好的，应为查找虚拟地址对应页表项时确实有可能不存在页表项，所以每次都会有一个缺失检查，如果缺失则会调用相应的创建函数、本质上仍旧是合在一起的。

练习3：给未被映射的地址映射上物理页（需要编程）

补充完成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。

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。

首先，根据给定的逻辑地址addr，在当前进程的内存管理结构mm中找到对应的VMA（虚拟内存区域）。

```
//try to find a vma which include addr
struct vma_struct *vma = find_vma(mm, addr);
```

判断VMA是否存在以及addr是否在VMA的范围内，如果不满足，则输出错误信息并返回。

```
//If the addr is in the range of a mm's vma?
if (vma == NULL || vma->vm_start > addr) {
    cprintf("not valid addr %x, and  can not find it in vma\n", addr);
    goto failed;
}
```

根据VMA的权限信息，构造所需的访问权限perm。该值将会在后续的page_insert()函数中被使用。vm_flags表示一段虚拟地址对应的权限（可读、可写、可执行等）。

```
/* IF (write an existed addr ) OR
 *    (write an non_existed addr && addr is writable) OR
 *    (read  an non_existed addr && addr is readable)
 * THEN
 *    continue process
 */
uint32_t perm = PTE_U;
if (vma->vm_flags & VM_WRITE) {
    perm |= (PTE_R | PTE_W);
}
```

将addr按页大小PGSIZE进行向下对齐。

```
addr = ROUNDDOWN(addr, PGSIZE);
```

尝试获取页表项pte，如果下一级页表不存在则分配一个物理页来建立新页表，并建立页表项以及逻辑地址addr的映射关系。

```
ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                        //PT(Page Table) isn't existed, then
                                        //create a PT.
```

如果pte已经存在，说明该页表项是一个交换条目，需要从磁盘加载数据到物理页，并建立映射关系，并设置页面可交换。

```
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {
        /*LAB3 EXERCISE 3: YOUR CODE
        * 请你根据以下信息提示，补充函数
        * 现在我们认为pte是一个交换条目，那我们应该从磁盘加载数据并放到带有phy addr的页面，
        * 并将phy addr与逻辑addr映射，触发交换管理器记录该页面的访问情况
        *
        *  一些有用的宏和定义，可能会对你接下来代码的编写产生帮助(显然是有帮助的)
        *  宏或函数:
        *    swap_in(mm, addr, &page) : 分配一个内存页，然后根据
        *    PTE中的swap条目的addr，找到磁盘页的地址，将磁盘页的内容读入这个内存页
        *    page_insert :  建立一个Page的phy addr与线性addr la的映射
        *    swap_map_swappable :  设置页面可交换
        */
        if (swap_init_ok) {
            struct Page *page = NULL;
            // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
            //(1) According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            swap_in(mm, addr, &page);
            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            page_insert(mm->pgdir, page, addr, perm);
```

```
            //(3) make the page swappable.
            swap_map_swappable(mm, addr, page, 1);

            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
    }
```

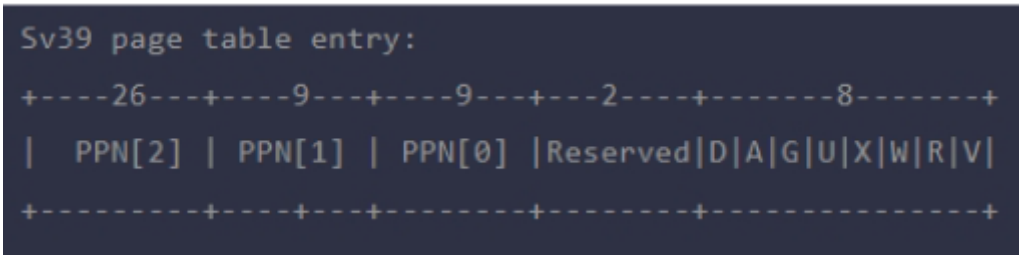
以上代码中，主要有三个步骤是关键的：页面换入，建立映射，属性设置。这三个步骤在练习1中已经说明，此处不再赘述。

如果交换管理器初始化失败，则输出错误信息并返回。

返回0表示成功完成。

潜在用处

页目录项（Page Directory Entry）和页表项（Page Table Entry）中的组成部分可以用于实现页替换算法。在sv39中，页表项的结构如下：



其中的字段可以分为两部分：用于查找页号的字段和用于权限设置的字段。

其中，关于权限设置的字段可以记录页面的访问情况、页面的修改状态、页面的使用频率等信息，从而为页替换算法提供更多的依据和参考。比如，do_pgfault中，就利用了页表项的权限字段。具体来说，在设置perm的值的时候，就使用到了用于权限设置的字段。

- 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？

  代码层面：如果出现页面访问异常，程序会进入`trap/trapentry.S`当中，保存上下文，再进入`trap.c`文件当中进行异常处理，处理结束后，再逐步退回至发生异常的代码处，重新执行。

  在`trap.c`文件当中：会从`pgfault_handler`函数中执行`do_pgfault`函数。

  在`pg_fault`函数当中，根据发生页面访问异常的地址`addr`寻找到对应的`vma`。获取`vma`的权限后，根据`addr`对齐后的地址，寻找相应的`page_table_entry`。如果没有找到，就立马创建一个`pte`。找到`pte`后，我们将该页面换入，存入物理地址和虚拟地址的映射，将该页面设置为可被替换的。

  当ucore的缺页服务例程在执行过程中访问内存出现页访问异常时，硬件会执行以下操作：

  - 将异常类型设置为页访问异常
  - 保存当前的程序状态/上下文（如PC、寄存器等）
  - 转移控制权到操作系统的缺页异常处理程序
  - 恢复上下文
  - 继续执行当前缺页异常处理程序
- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？
  - 当然是与关系的。

- 我们看Page结构体的组成：

```
struct Page {
    int ref;
    uint_t flags;
    uint_t visited;
    unsigned int property;
    list_entry_t page_link;
    list_entry_t pra_page_link;
    uintptr_t pra_vaddr;
};
```

其中的pra_vaddr就是这个页所对应的虚拟地址。根据这个地址，我们就可以知道其对应的页目录项和页表项。Page数组的索引（表示一个内存页）对应页表项的索引，而Page数组的每一项中存储了该页表项对应的物理页的相关信息，比如物理地址、引用计数、是否在交换区等。这样可以通过Page数组来管理和跟踪物理页的状态和使用情况。

练习4：补充完成Clock页替换算法（需要编程）

通过之前的练习，相信大家对FIFO的页面替换算法有了更深入的了解，现在请在我们给出的框架上，填写代码，实现 Clock页替换算法（mm/swap_clock.c）。(提示:要输出curr_ptr的值才能通过make grade)

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：

在初始化函数`_clock_init_mm(struct mm_struct *mm)`中：

- 初始化`pra_list_head`为空链表
- `curr_ptr`初始化为链表头
- `mm->sm_priv`初始化为链表头，用于后续的页面替换操作

在`_clock_map_swappable(……)`函数当中：将page对应的 entry 加到 pra_list 的队尾，并将该页初始化为visited=1.

```
    list_entry_t *head=mm->sm_priv;
    //注意这里！！！因为后面的victim函数中把curr_ptr放到了victim的后面了
    list_add(list_prev(curr_ptr),entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited=1;
```

在`_clock_swap_out_victim(……)`函数当中：

- 利用`curr_ptr`指针，遍历整个`pra_list`。
    - 如果遇到某个页面visited=0，则该页就是被换出的受害页
    - 如果某个页面visited=1，则将其置为0；继续往下查找

在整个的while循环当中：

```
while (1) {
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 编写代码
        // 遍历页面链表pra_list_head，查找最早未被访问的页面
        // 获取当前页面对应的Page结构指针
        // 如果当前页面未被访问，则将该页面从页面链表中删除，并将该页面指针赋值给ptr_page作为换出页面
        // 如果当前页面已被访问，则将visited标志置为0，表示该页面已被重新访问
```

```
            if(curr_ptr==head)
        {
            curr_ptr=curr_ptr->next;
        }
        // 获取当前页面对应的Page结构指针
         victim=le2page(curr_ptr,pra_page_link);
        // 如果当前页面未被访问，则将该页面从页面链表中删除，并将该页面指针赋值给ptr_page作为
换出页面
        if(victim->visited==0)
        {
            list_entry_t * t=curr_ptr;
            curr_ptr=curr_ptr->next;
            *ptr_page=victim;

            cprintf("curr_ptr 0xffffffff%08x\n",(uintptr_t)t);
            list_del(t);
            break;
        }
        // 如果当前页面已被访问，则将visited标志置为0，表示该页面已被重新访问
        victim->visited=0;
        curr_ptr=curr_ptr->next;
    }
```

- 比较Clock页替换算法和FIFO算法的不同。
  - clock算法主体是通过curr_ptr 时钟指针循环遍历整个链表，如果在上一轮中某页未被访问过，就要将其选为victim。FIFO算法并不关心某页有没有被访问过，只是依靠页面进入的先后顺序（先进先出的原则）选择victim。
  - 相较而言，clock算法需要循环遍历链表，最差的情况时，需要循环一轮整个链表；而FIFO只需要从队列出口取出一页即可。因此，Clock的算法时间复杂度相较高一些。

练习5：阅读代码和实现手册，理解页表映射方式相关知识（思考题）

如果我们采用”一个大页“ 的页表映射方式，相比分级页表，有什么好处、优势，有什么坏处、风险？

- 一级页表：
  - 优点
    - 实现简单，只需要维护一个表即可！
    - 访问速度快，直接访问页表即可，无需其他的查找操作
    - 只需要一个页表，占用内存少。
  - 缺点
    - 如果虚拟内存非常大，页表所需要的物理内存也会变大。
    - 如果页表中的页表项数量增多，查找页表项就会很慢，访问相对而言就会变慢
- 分级页表
  - 优点
    - 适用于大内存空间，引入多级页表可以将页表项分散到多个页表中存储
  - 缺点
    - 需要维护多个表
    - 访问存时，需要访问多个表才能最终访问到物理地址耗时
    - 如果页表项较少，不太适用分级页表，反而会占用更多的内存。

————————————

对于我们的sv39的虚拟地址来说，除去页内偏移12位：

- 如果采用一级页表
  - 需要些2^(39-12)个页表，存储这些页表需要2^27*8B
- 如果采用三级页表
  - 每一级需要2((39-12)/3)个页表，存储这些页表需要 2^9*3*8B 的大小，内存消耗少了很多，但是，寻址时需要访问多个地址，时间消耗增加

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

challenge部分不是必做部分，不过在正确最后会酌情加分。需写出有详细的设计、分析和测试的实验报告。完成出色的可获得适当加分。

【见分册】

二、实验概述

- 管理虚拟内存
  - 初始化虚拟内存管理机制
    - 设置好页面是存在物理内存中还是磁盘中
    - 完善建立页表映射、页面访存异常
  - 访存测试
    - 建立的页表项是否能够正确完成虚实映射
    - 建立的页表项是否正确庙湖虚拟内存页在物理内存上还是在硬盘上
    - 虚拟内存页在内存和硬盘上的正确传递
    - 页面替换算法
- 实验流程
  - init初始化
    - pmm_init=>物理内存管理的初始化
    - pic_init,idt_init =>初始化处理器中断控制器pic， 中断描述符表IDT
    - vmm_init 虚拟内存管理机制的初始化
      - 建立虚拟内存和物理内存映射关系
    - ide_init =>swap硬盘的初始化工作
    - 准备好硬盘读写工作
    - swap_init =>初始化页面置换算法
- qemu 与数据结构
  - qemu模拟出来的ram 看做硬盘
  - ide 硬盘接口
    - ide_read_secs
    - ide_write_secs
  - fs 硬盘和内核之间的接口
    - swapfs_init
    - swapfs_read
    - swapfs_write

三、知识点总结

多级页表实现虚拟存储

- 修改页表
  - page_insert
    - 获取虚拟地址对应的页表项的位置ptep
  - page_remove
  - page_ref ，一个物理页面被多少虚拟页面所对应

虚拟内存

CPUor程序员看到的内存，并不一定对应物理内存单元。

内存地址虚拟化的两大用处：

- 设置页表限定软件运行时的访问空间
- 软件访问时才动态分配物理内存；页面的换入换出将不经常访问的数据放在磁盘上，节省内存空间

Page Fault

- CPU发射虚拟地址，经过MMU转换成物理地址，再从地址总线发出才能访问地址总线上的外设。因此，需要考虑虚拟地址和物理地址之间的转换关系与物理页访问权限。
- 只有用户程序可以直接访问的页用于换出，内核空间的页不可以被换出
- 处理缺页异常：
  - 在`kern/trap/trap.c`的`exception_handler()`，按照`scause`寄存器对异常的分类里，有`CAUSE_LOAD_PAGE_FAULT`和`CAUSE_STORE_PAGE_FAULT`两个case。这里的异常处理程序会把Page Fault分发给`kern/mm/vmm.c`的`do_pgfault()`函数并尝试进行页面置换。

虚拟内存的管理

虚拟内存单元属性

（1）虚拟内存不一定有对应的物理内存，且二者地址通常不相等；

（2）操作系统实现的某种内存映射可建立虚拟内存到物理内存的对应关系。

使用多级页表实现虚拟存储

（1）页表项

（2）定义页表相关操作（在pmm.c中）：建立、新增、删除映射关系。

注：Lab2中多级页表是在entry.S中定义并实现的，本实验中则是新定义在了一个文件中。前者是比较粗糙的（各段权限相同），后者则是选择放弃前者新建页表、并让satp指向该表。

4、缺页异常

（1）哪些页可以被换出——映射至用户空间并被用户程序直接访问的页面

（2）do_pgfault——位于vmm.c中

按需分页

`vma_struct`:一段连续的互不重合的虚拟地址，从`vm_start`到`vm_end`。 通过包含一个`list_entry_t`成员，我们可以把同一个页表对应的多个`vma_struct`结构体串成一个链表，在链表里把它们按照区间的起始点进行排序。

`vm_flags`表示的是一段虚拟地址对应的权限（可读，可写，可执行等），这个权限在页表项里也要进行对应的设置。

页换入换出（页面置换）

在do_pgfault函数当中，执行swap_in,swap_map_swappable等函数，这些函数底层调用的就是clock算法或FIFO算法等。

```
//do_pgfault():
if (swap_init_ok) {
            struct Page *page = NULL;
            //在swap_in()函数执行完之后，page保存换入的物理页面。
            //swap_in()函数里面可能把内存里原有的页面换出去
            swap_in(mm, addr, &page);  //(1) According to the mm AND addr, try
                                       //to load the content of right disk page
                                       //into the memory which page managed.
            page_insert(mm->pgdir, page, addr, perm); //更新页表，插入新的页表项
            //(2) According to the mm, addr AND page,
            // setup the map of phy addr <---> virtual addr
            swap_map_swappable(mm, addr, page, 1);  //(3) make the page
swappable.
            //标记这个页面将来是可以再换出的
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
```

FIFO

该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象（Belady 现象），即在增加放置页的物理页帧的情况下，反而使页访问异常次数增多。

LRU

用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。

CLOCK

是 LRU 算法的一种近似实现。时钟页替换算法把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针（简称当前指针）指向最老的那个页面，即最先进来的那个页面。另外，时钟算法需要在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当操作系统需要淘汰页时，对当前指针指向的页所对应的

页表项进行查询，如果访问位为“0”，则淘汰该页，如果该页被写过，则还要把它换出到硬盘上；如果访问位为“1”，则将该页表项的此位置“0”，继续访问下一个页。该算法近似地体现了 LRU 的思想，且易于实现，开销少，需要硬件支持来设置访问位。时钟页替换算法在本质上与 FIFO 算法是类似的，不同之处是在时钟页替换算法中跳过了访问位为 1 的页。

Enhanced Clock

页替换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面，所以优先淘汰没有修改的页，减少磁盘操作次数。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。