lab3 虚拟内存管理

实验目的：

- 了解虚拟内存的PageFault异常处理实现，
- 了解页替换算法在操作系统中的作用

练习

练习0：

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。

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。

pmm.init():物理内存的管理

vmm.init():虚拟内存的管理

vma_struct（vmm.h）

```
struct vma_struct {
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    uintptr_t vm_start;      // start addr of vma
    uintptr_t vm_end;        // end addr of vma, not include the vm_end itself
    uint32_t vm_flags;       // flags of vma
    list_entry_t list_link;  // linear list link which sorted by start addr of
vma
};
/*
vm_start与vm_end为一片连续的虚拟空间的起始，按页对齐
list_link为连接起所有vma对象的指针
vm_flags为虚拟内存空间的属性：
#define VM_READ                 0x00000001      只读
#define VM_WRITE                0x00000002      读写
#define VM_EXEC                 0x00000004      可执行
其中还包含一个mm_struct的对象

// the control struct for a set of vma using the same PDT
struct mm_struct {
    list_entry_t mmap_list;        // 链接所有属于同一页目录表的虚拟空间
    struct vma_struct *mmap_cache; // 当前正在使用的虚拟内存空间
    pde_t *pgdir;                  // 所维护的页表
    int map_count;                 // 虚拟空间的个数
    void *sm_priv;                 // 链接记录页访问情况的链表头
};
*/
```

```
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    //参数：内存总管对象mm，错误码error_code，具体出错的线性地址
    int ret = -E_INVAL;

    //尝试找到这一地址对应的vma
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    //这一地址不是有效的虚拟内存空间，报错
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    //check the error_code
    //检查错误码
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        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: /* error code flag : (W/R=0, P=1): read, present */
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        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;
        }
    }
    /* 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_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    /*LAB3 EXERCISE 1: YOUR CODE
    * Maybe you want help comment, BELOW comments can help you finish the code
```

```
    *
    * Some Useful MACROs and DEFINEs, you can use them in below implementation.
    * 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
    * DEFINES:
    *   VM_WRITE  : If vma->vm_flags & VM_WRITE == 1/0, then the vma is
writable/non writable
    *   PTE_W           0x002                   // page table/directory entry
flags bit : Writeable
    *   PTE_U           0x004                   // page table/directory entry
flags bit : User can access
    * VARIABLES:
    *   mm->pgdir : the PDT of these vma
    *
    */
#if 0
    /*LAB3 EXERCISE 1: YOUR CODE*/
    ptep = ???              //(1) try to find a pte, if pte's PT(Page Table)
isn't existed, then create a PT.
    if (*ptep == 0) {
                            //(2) if the phy addr isn't exist, then alloc a page
& map the phy addr with logical addr

    }
    else {
    /*LAB3 EXERCISE 2: YOUR CODE
    * Now we think this pte is a  swap entry, we should load data from disk to a
page with phy addr,
    * and map the phy addr with logical addr, trigger swap manager to record the
access situation of this page.
    *
    *  Some Useful MACROs and DEFINEs, you can use them in below implementation.
    *  MACROs or Functions:
    *    swap_in(mm, addr, &page) : alloc a memory page, then according to the
swap entry in PTE for addr,
    *                               find the addr of disk page, read the content
of disk page into this memroy page
    *    page_insert :  build the map of phy addr of an Page with the linear addr
la
    *    swap_map_swappable :  set the page swappable
    */
        if(swap_init_ok) {
            struct Page *page=NULL;
                                    //(1) According to the mm AND addr, try to
load the content of right disk page
                                    //    into the memory which page managed.
                                    //(2) According to the mm, addr AND page,
setup the map of phy addr <---> logical addr
                                    //(3) make the page swappable.
```

```
            }
            else {
                cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
                goto failed;
            }
        }
#endif
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }

    if (*ptep == 0) {  //二级页表入口
        // 令pgdir指向的页表中，la线性地址对应的二级页表项与一个新分配的物理页Page进行虚实地址的映射
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else {  //页表项非空，尝试换入页面
        if(swap_init_ok) {
            // 如果开启了swap磁盘虚拟内存交换机制
            struct Page *page=NULL;
            // 将addr线性地址对应的物理页数据从磁盘交换到物理内存中(令Page指针指向交换成功后的物理页)
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            page_insert(mm->pgdir, page, addr, perm);   //建立虚拟地址和物理地址之间的对应关系，perm设置物理页权限，为了保证和它对应的虚拟页权限一致
            swap_map_swappable(mm, addr, page, 1);  //也添加到算法所维护的次序队列
            page->pra_vaddr = addr;     //设置页对应的虚拟地址
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
   ret = 0;
failed:
    return ret;
}
```

一个应用进程，试图对某一地址上的内容进行操作。该地址为一个虚拟地址，OS需要寻找到这一虚拟地址与实际的物理地址的映射关系。但出现了问题，可能是因为当前并不存在这样的映射关系，也可能是试图对这段地址进行写入，但管理这段地址的vmm表明这个地址并不支持写入。发生了错误，因此需要进行处理

在处理时，给出了如下的信息，内存总管对象mm，错误码errorcode，和访问出现问题的线性地址。首先检查这个地址是否合法（vmm对象是否存在，地址是否超出了vmm的范围）不合法直接免谈。

然后检查错误类型，能做到的异常处理仅仅是把一个真的可以访问的东西从磁盘中挪到内存中。错误码共四种情况

11：应用进程想写这段空间，这段空间也确实在内存里

10：应用进程想写这段空间，这段空间不在内存里

以上两种情况一起处理。通过检查vmm对象，如果允许写的话，说明后续还可以通过把页换进来解决问题，如果不允许写的话。洗洗睡

01：应用进程想读，这段空间在内存里，就是不让读，直接寄

00：应用进程想读，这段空间不在内存里，检查一下vmm的read和exec，有一个能用就行，

通过如上考验，当前错误还有抢救的可能，开始准备给这个出问题的线性地址分配一个物理页（give this man a shield）

准备一下标志位perm，肯定得让用户能访问到，初始给一个PTE_U，如果虚拟地址允许写的话，物理地址的状态位也跟一个PTE_W，addr按页对齐一下，一会要作为页的起始地址去获得页表项的。

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
pde_t *pdep = &pgdir[PDX(la)];
    if (!(*pdep & PTE_P)) {
        //不存在对应的页表
        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);
        *pdep = pa | PTE_U | PTE_W | PTE_P;
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

切换到lab2的思路上，现在我们有了一个虚拟地址，想找到他对应的物理页，首先高十位作为页目录表的索引，获得一个页目录项pdep,这个页目录项有效的话，以线性地址中间十位作为二级页表中的索引，获得页表项，**（只是页表项，其中的数据才是页的地址）**因为之前已经把addr按页对齐过了，因此也就不存在什么12位的偏移了，结束战斗。

还有一种情况，这个页目录项无效。有点搞。这个时候给他分配一个页作为二级页表。分配了一个页，设置了引用次数，把这个页的地址连同一些标志位，可访问，可写，有效，作为页目录表的一个页目录项。

回到do_pgfault上，不出以外现在已经获取到了一个二级页表项了，如果这是一个新创建的页，*ptep就会是0（啥也没存过），此时二级页表项中的内容只是一个内存地址，不能真正称为是一个页，因为没有page结构在管理它。通过pgdir_alloc_page进行一下初始化，

```
struct Page *
pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page();
    if (page != NULL) {
        // 建立la对应二级页表项(位于pgdir页表中)与page物理页基址的映射关系
        if (page_insert(pgdir, page, la, perm) != 0) {
            free_page(page);
            return NULL;
        }
```

```
        if (swap_init_ok){
            // 将新映射的这一个page物理页设置为可交换的，并纳入全局swap交换管理器中管理
            swap_map_swappable(check_mm_struct, la, page, 0);
            // 设置这一物理页关联的虚拟内存
            page->pra_vaddr=la;
            assert(page_ref(page) == 1);
            //cprintf("get No. %d  page: pra_vaddr %x, pra_link.prev %x,
pra_link_next %x in pgdir_alloc_page\n", (page-pages), page->pra_vaddr,page-
>pra_page_link.prev, page->pra_page_link.next);
        }

    }
    return page;
}
```

里面又涉及到了page_insert这个函数

```
int
page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    /*
    参数位一个页目录表，一个刚分配出来的页，一个线性地址，一列标志位
    */
    //获取了二级页表项（刚弄来的）
    pte_t *ptep = get_pte(pgdir, la, 1);
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    //把page引用了一次
    page_ref_inc(page);
    if (*ptep & PTE_P) {
        //挺关键的，这里pte2page中式ptep的内容，取了二级页表项的内容，即页的基址
        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;
    //应该是因为更新了映射关系，所以要把tlb中的原映射清除吧。
    tlb_invalidate(pgdir, la);
    return 0;
}
```

对于do_pagefault->getpte->pgdir_alloc_page->page_insert这一条路来说，就是对没有物理地址进行映射的线性地址，分配一个页的空间，这个页本身还有一个page结构在管理，通过在页目录表中和二级页表中进行了完善，

现在剩下的问题是pgdir_alloc_page中有关swap的部分，do_pagefault中剩下的也是一部分swap的内容，先放放。

从swap_init_ok入手，这是一个标志位，代表了swap模块是否初始化完成，虽然暂时还不清楚什么时候调用了swap进行了初始化，但还是先看看swap_init干了什么。

```
int
swap_init(void)
{
    //又初始化了一个东西
     swapfs_init();

     if (!(1024 <= max_swap_offset && max_swap_offset < MAX_SWAP_OFFSET_LIMIT))
     {
          panic("bad max_swap_offset %08x.\n", max_swap_offset);
     }


     sm = &swap_manager_fifo;
     int r = sm->init();

     if (r == 0)
     {
          swap_init_ok = 1;
          cprintf("SWAP: manager = %s\n", sm->name);
          check_swap();
     }

     return r;
}
```

```
//不太懂，先放放
void
swapfs_init(void) {
    static_assert((PGSIZE % SECTSIZE) == 0);
    if (!ide_device_valid(SWAP_DEV_NO)) {
        panic("swap fs isn't available.\n");
    }
    max_swap_offset = ide_device_size(SWAP_DEV_NO) / (PGSIZE / SECTSIZE);
}
```

大致意思是完成了内存交换模块的一个初始化，嗯，应该是，

回到do_pgfault吧，反正如果swap模块成功初始化了，就新声明了一个页，调用了swap_in函数，输入参数为内存总管mm，线性地址和那个空的page指针，

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
    //分配了一个页
     struct Page *result = alloc_page();
     assert(result!=NULL);
    //拿到了这个线性地址对应的二级页表项
     pte_t *ptep = get_pte(mm->pgdir, addr, 0);
     // cprintf("SWAP: load ptep %x swap entry %d to vaddr 0x%08x, page %x, No
%d\n", ptep, (*ptep)>>8, addr, result, (result-pages));

     int r;
    //这里是是取了二级页表项的内容，即这个线性地址对应的物理页的基址，调用了swapfs_read
    //应该是将硬盘中线性地址对应的物理页交换到了result中，最终将result返回
```

```
     if ((r = swapfs_read((*ptep), result)) != 0)
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n",
(*ptep)>>8, addr);
     *ptr_result=result;
     return 0;
}
```

```
int
swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

随后建立起了建立虚拟地址和物理地址之间的对应关系，perm设置物理页权限，为了保证和它对应的虚拟页权限一致，将这个地址和页添加到算法所维护的次序队列，do_pgfault结束。

练习2：补充完成基于FIFO的页面替换算法（需要编程）

使用FIFO策略的换入换出

先进先出(First In First Out, FIFO)页替换算法：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。
page结构有所调整

```
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)
};
```

前面的属性都和之前没啥区别，问题在最后两个变量上，pra_vaddr好理解，就是用以记录这个page结构对应的线性地址，pra_page_link，用于记录加入内存顺序这方面的内容。pra_page_link可用来构造按页的第一次访问时间进行排序的一个链表，这个链表的开始表示第一次访问时间最近的页，链表结尾表示第一次访问时间最远的页。

FIFO的初始化

```
static int
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

主要内容是初始化用以记录页访问顺序的双向链表，链表头为pra_list_head，同时把它传给内存总管mm

```
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*/
    //(1)link the most recent arrival page at the back of the pra_list_head
qeueue.
    list_add(head, entry);
    return 0;
}
```

swappable函数，将当前换进来的页加入到mm对象的页访问顺序链表中，每次加在链表头的后面，这样越先被访问的页，在链表中越靠后，

```
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*/
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  assign the value of *ptr_page to the addr of this page
     list_entry_t *le = head->prev;
     assert(head!=le);
     struct Page *p = le2page(le, pra_page_link);
     list_del(le);
     assert(p !=NULL);
     *ptr_page = p;
     return 0;
}
```

准备换出的函数，in_tick是在检查什么还不太清楚，反正函数的主要功能是找到最先进来的页，在链表里把它删掉。同时把对应的page对象准备换出去。

补充一些细碎的点

每一个vma都代表了一个进程的合法虚拟内存空间，

每一个mm作为内存管理器，统一管理一个进程的虚拟内存以及物理内存，

换入与换出：

换出：

积极：每隔一段时间主动检测长时间未使用的页，将其换出到硬盘之中，保证任何时刻内存中总有空闲空间

消极：发生换入时，找一个倒霉蛋换出去

换入：

当想找一个线性地址对应的物理地址映射，但这个映射此时不在内存之中，发生一次换入

页错误

页错误分两种情况，一种是直接发现缺页了，此时发生缺页错误，还有抢救的可能，可以试图把缺失的页换到内存中，另一种是发现权限有问题，不让读或者不让写，那就没救了

扩展练习

challenge1：实现识别dirty bit的 extended clock页替换算法（需要编程）

页替换算法由swap.c中的swap_manager对象来控制，可以主动实现不同的页替换算法，在swap_init中挂接到sm对象上

```
struct swap_manager swap_manager_clock =
{
     .name            = "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,
};
```

实现改进版的时钟算法，swap_clock,其头文件swap_clock.h的设计可仿照swap_fifo.h进行。重点在于swap_clock.c中各个函数的实现。

```
//初始化，将mm管理的访问序列置空
static int
_clock_init_mm(struct mm_struct *mm)
{
     mm->sm_priv = NULL;
     return 0;
}
```

_clock_init_mm：实现初始化操作，将当前mm对象维护的访问队列置空，

```
//维护访问队列，
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
int swap_in)
{
    //链表头
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    //要插入的页
    list_entry_t *entry=&(page->pra_page_link);
    assert(entry != NULL);

    // Insert before pointer
    if (head == NULL) {
        list_init(entry);
        mm->sm_priv = entry;
    } else {
        //越新进来的页越靠后
        list_add_before(head, entry);
    }
    return 0;
}
```

_clock_map_swappable：访问队列的维护，在创建新的映射关系或者发生换入时需要将新的页添加到访问队列中。通过list_add_before将后产生的页放置在链表中靠后的地方。

```
//选择倒霉蛋
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int
in_tick)
{
    //链表入口
     list_entry_t *head=(list_entry_t*) mm->sm_priv;
     assert(head != NULL);
     assert(in_tick==0);

     list_entry_t *selected = NULL, *p = head;
     // Search <0,0>
     do {
        //
        if (GET_ACCESSED_FLAG(mm->pgdir, p) == 0 && GET_DIRTY_FLAG(mm->pgdir, p)
== 0) {
            selected = p;
            break;
        }
        p = list_next(p);
     } while (p != head);
     // Search <0,1> and set 'accessed' to 0
     if (selected == NULL)
        do {
            if (GET_ACCESSED_FLAG(mm->pgdir, p) == 0 && GET_DIRTY_FLAG(mm-
>pgdir, p)) {
                CLEAR_ACCESSED_FLAG(mm->pgdir, p);
                selected = p;
                break;
```

```
            }
            CLEAR_ACCESSED_FLAG(mm->pgdir, p);
            p = list_next(p);
        } while (p != head);
    // Search <0,0> again
    if (selected == NULL)
        do {
            if (GET_ACCESSED_FLAG(mm->pgdir, p) == 0 && GET_DIRTY_FLAG(mm-
>pgdir, p) == 0) {
                selected = p;
                break;
            }
            p = list_next(p);
        } while (p != head);
    // Search <0,1> again
    if (selected == NULL)
        do {
            if (GET_ACCESSED_FLAG(mm->pgdir, p) == 0 && GET_DIRTY_FLAG(mm-
>pgdir, p)) {
                selected = p;
                break;
            }
            p = list_next(p);
        } while (p != head);
    // Remove pointed element
    head = selected;
    if (list_empty(head)) {
        mm->sm_priv = NULL;
    } else {
         mm->sm_priv = list_next(head);
        list_del(head);
    }
    *ptr_page = le2page(head, pra_page_link);
    return 0;
}
```

_clock_swap_out_victim：换出策略的实现，当需要进行换出操作时，通过此函数选择出换出哪一个页

时钟页替换算法：

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。

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。

改进的时钟页替换算法需要对访问队列进行多次遍历，在第一次遍历的过程中，主要寻找Access为0，Dirty为0的页，即在最近没有被访问过，同时也没有被修改过。这是最理想的情况。当不存在这样的页时，会进行第二次遍历，第二次遍历中，主要负责寻找Dirty为1，Access为0的页，即最近没有被访问过，但之前被修改过。这样的页是需要进行一个换出操作的，即一次额外的磁盘访问。但此时进行完整的换入换出必定会对这个页进行一次换出，因此选择这样的页并不会产生额外开销。第二次遍历的另一个任务是将遇到的Access为1的页Access置为0，当第二次访问的时候仍没有得到结果时，会进行第三次访问，同样寻找Access为0，Dirty为1的页。经过第二次遍历后，原本Access为1的页都被置为了0.因此第三次遍历也相当于在寻找原本Access为1，Dirty为0的页。当第三次遍历时仍没有结果时，会进行第四次遍历，此时寻找的就是最坏的情况， 即Access为1（原始情况），Dirty也为1的页。

swap_clock.c中的其他函数可保持不变。