# Lab13 页面置换

## 一、 实验概述

在本次实验中，我们将实现页面置换算法。

## 二、实验目的

1. 掌握页面置换的原理
2. 掌握页面置换算法

## 三、实验项目整体框架概述

// Lab13（同Lab11)

```
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── consh
│   │   ├── ide.c//模拟硬盘
│   │   ├── ide.h
│   │   ├── intr.c
│   │   └── intr.h
│   ├── fs //模拟硬盘
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   │   ├── default_pmm.c
│   │   ├── default_pmm.h
│   │   ├── memlayout.h
│   │   ├── mmu.h
│   │   ├── pmm.c //更新了分配函数
│   │   ├── pmm.h
│   │   ├── swap.c//替换的相关实现
│   │   ├── swap_fifo.c //fifo 替换算法
│   │   ├── swap_fifo.h
│   │   ├── swap.h
│   │   ├── vmm.c  //虚拟内存管理相关信息
│   │   └── vmm.h
│   ├── sync
│   └── trap
```

```
│   ├── trap.c
│   ├── trap.h
│   └── trapentry.S
├── libs
├── Makefile
└── tools
```

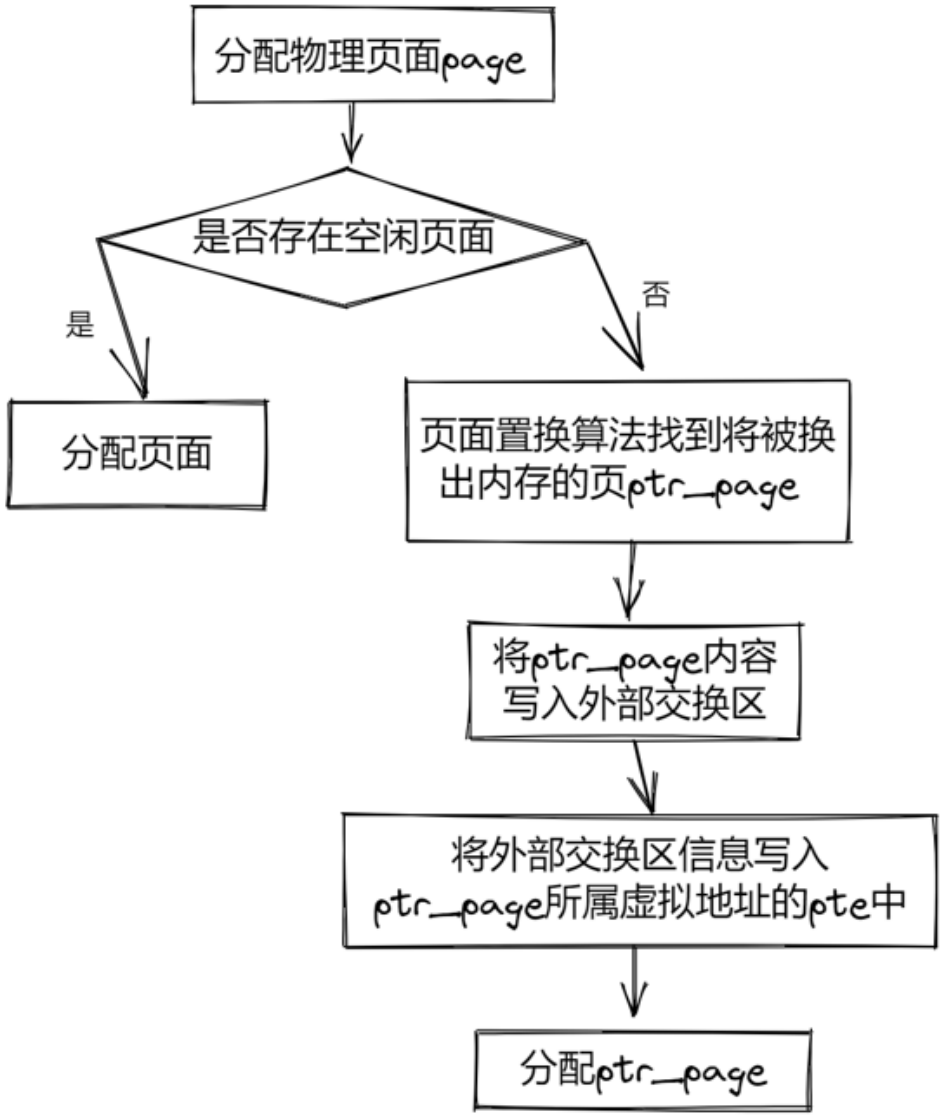
# 四、实验内容

1. 学习页面置换算法并进行实现

# 五、实验过程概述及相关知识点

## 1. 分配页面，页面置换

当需要在物理内存分配页面时，我们需要调用之前物理内存管理实验中分配页面的方法。但是在之前的实验代码中，我们认为是存在空闲页面的，因而未处理过没有空闲页面的情况。**当没有空闲页面，又需要分配新的页面时，则会需要进行页面置换**，即从现有页面中选出一页，将其内容放入交换区，再把该页面分配给进程。具体流程如下：




> **页面置换算法**

> 从当前页面选择被置换出的具体页面时，我们希望把即将使用的页面（常用）尽量保留，即尽量选择最近不会被使用的页面（不常用），从而减少page fault的产生，因而产生了各种算法用于选择页面：
>
> - 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。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。
> - 最久未使用(least recently used, LRU)算法：利用局部性，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。
> - 时钟（Clock）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。
> - 改进的时钟（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。

本次实验代码会实现页面置换算法中的fifo算法。

# 六、代码实现

## 1.页面置换

#### 换入换出的具体实现

换入的时候，我们新分配一个物理页面，根据虚拟地址对应的pte在硬盘上找到对应的位置（这里用的是简单的偏移实现），将硬盘的内容换入到新分配的页面。

```
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();//这里alloc_page()内部可能调用swap_out()
```

```
     //找到对应的一个物理页面
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);//找到/构建对应的页表项
    //将物理地址映射到虚拟地址是在swap_in()退出之后，调用page_insert()完成的
     int r;
     if ((r = swapfs_read((*ptep), result)) != 0)//将数据从硬盘读到内存
     {
        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;
}
```

换出的时候，首先根据替换算法找到要被换出的那一个page，称为victim。接着找到victim对应的虚拟地址，根据这个虚拟地址构造一个可以放在硬盘上的位置，将victim换出。同时在更改victim对应的页表项，将硬盘对应的位置放入到victim的页表项中，最后释放掉对应的物理页面。

```
int swap_out(struct mm_struct *mm, int n, int in_tick)
{
     int i;
     for (i = 0; i != n; ++ i)
     {
          uintptr_t v;
          struct Page *page;
          int r = sm->swap_out_victim(mm, &page, in_tick);//调用页面置换算法的接口
          //r=0表示成功找到了可以换出去的页面
         //要换出去的物理页面存在page里
          if (r != 0) {
                  cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                  break;
          }

          cprintf("SWAP: choose victim page 0x%08x\n", page);

          v=page->pra_vaddr;//可以获取物理页面对应的虚拟地址
          pte_t *ptep = get_pte(mm->pgdir, v, 0);
          assert((*ptep & PTE_V) != 0);

          if (swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page) != 0) {
                      //尝试把要换出的物理页面写到硬盘上的交换区，返回值不为0说明失败了
                  cprintf("SWAP: failed to save\n");
                  sm->map_swappable(mm, v, page, 0);
                  continue;
          }
          else {
              //成功换出
                  cprintf("swap_out: i %d, store page in vaddr 0x%x to disk
swap entry %d\n", i, v, page->pra_vaddr/PGSIZE+1);
                  *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
                  free_page(page);
          }
          //由于页表改变了，需要刷新TLB
          tlb_invalidate(mm->pgdir, v);
     }
```

```
    return i;
}
```

**替换管理器**

类似`pmm_manager`，我们定义`swap_manager`，组合页面置换需要的一些函数接口。

```
struct swap_manager
{
     const char *name;
     /* Global initialization for the swap manager */
     int (*init)            (void);
     /* Initialize the priv data inside mm_struct */
     int (*init_mm)         (struct mm_struct *mm);
     /* Called when tick interrupt occured */
     int (*tick_event)      (struct mm_struct *mm);
     /* Called when map a swappable page into the mm_struct */
     int (*map_swappable)   (struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in);
     /* When a page is marked as shared, this routine is called to
      * delete the addr entry from the swap manager */
     int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
     /* Try to swap out a page, return then victim */
     int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page, int
in_tick);
     /* check the page relpacement algorithm */
     int (*check_swap)(void);
};
```

在我们完成替换功能的初始化的时候，我们首先初始化我们的硬盘，设置对应替换算法的管理器，然后把替换的全局标志位swap_init_ok设置为1，这样标志着替换功能已经初始化完成。

```
// kern/mm/swap.c
static struct swap_manager *sm;
int swap_init(void)
{
     swapfs_init();

     // Since the IDE is faked, it can only store 7 pages at most to pass the
test
     if (!(7 <= max_swap_offset &&
        max_swap_offset < MAX_SWAP_OFFSET_LIMIT)) {
        panic("bad max_swap_offset %08x.\n", max_swap_offset);
     }

     sm = &swap_manager_fifo;//use first in first out Page Replacement Algorithm
     int r = sm->init();

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

     return r;
```

```
}

int swap_init_mm(struct mm_struct *mm)
{
     return sm->init_mm(mm);
}

int swap_tick_event(struct mm_struct *mm)
{
     return sm->tick_event(mm);
}

int swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
int swap_in)
{
     return sm->map_swappable(mm, addr, page, swap_in);
}

int swap_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
     return sm->set_unswappable(mm, addr);
}
```

**替换算法的实现**

这里我们以FIFO算法进行举例

FIFO(First in, First out)页面置换算法，就是把所有页面排在一个队列里，每次换入页面的时候，把队列里最靠前（最早被换入）的页面置换出去。`kern/mm/swap_fifo.c`完成了FIFO置换算法最终的具体实现。我们所做的就是维护了一个队列（用链表实现）。

_fifo_init_mm：初始化一个FIFO队列，并且将mm_struct中的sm_priv指向这个队列。

_fifo_map_swappable：将对应的page设置成为一个可以被替换的页面，即将page->pra_page_link的插入到FIFO队列。

_fifo_swap_out_victim：根据FIFO算法，选取要被换出的物理页面。

```
// kern/mm/swap_fifo.h
#ifndef __KERN_MM_SWAP_FIFO_H__
#define __KERN_MM_SWAP_FIFO_H__

#include <swap.h>
extern struct swap_manager swap_manager_fifo;

#endif
// kern/mm/swap_fifo.c
 /* In order to implement FIFO PRA, we should manage all swappable pages, so we
can link these pages into pra_list_head according the time order.
 */

list_entry_t pra_list_head;
/*
 *  _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr of
pra_list_head.
 *     Now, From the memory control struct mm_struct, we can access FIFO PRA
 */
static int
```

```
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     return 0;
}
/*
 * _fifo_map_swappable: According FIFO PRA, we should link the most recent
arrival page at the back of pra_list_head qeueue
 */
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
    //link the most recent arrival page at the back of the pra_list_head qeueue.
    list_add(head, entry);
    return 0;
}
/*
 *  _fifo_swap_out_victim: According FIFO PRA, we should unlink the  earliest
arrival page in front of pra_list_head qeueue,
 *    then set the addr of addr of this page to ptr_page.
 */
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 */
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  set the addr of addr of this page to ptr_page
    list_entry_t* entry = list_prev(head);
    if (entry != head) {
        list_del(entry);
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        *ptr_page = NULL;
    }
    return 0;
}

struct swap_manager swap_manager_fifo =
{
     .name            = "fifo swap manager",
     .init            = &_fifo_init,
     .init_mm         = &_fifo_init_mm,
     .tick_event      = &_fifo_tick_event,
     .map_swappable   = &_fifo_map_swappable,
     .set_unswappable = &_fifo_set_unswappable,
     .swap_out_victim = &_fifo_swap_out_victim,
     .check_swap      = &_fifo_check_swap,
```

```
};
```

## 2.测试函数简介

`check_swap`和`_fifo_check_swap`：在这两个测试函数中，我们首先分配四个物理页面，然后把free_list清空，即我们只有四个物理页面。有效的虚拟地址为[0x1000,0x6000)，对应于5个虚拟页面，这样就会产生硬缺页中断。我们按照一定的顺序分别使用五个虚拟页面的地址，检查FIFO算法的正确性。

# 七、本节知识点回顾

## 在本次实验中，你需要了解以下知识点：

1. 简单页面置换算法的实现

# 八、下一实验简单介绍

下一步，我们将进入文件系统的部分。