操作系统实验报告

实验名称：缺页异常和页面置换

学号：2211349 2212126 2213603

小组成员：周末 周思洁 李娅琦

1.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

下面是在缺页时进行页面交换的过程中对经过的函数/宏的解释:

- **do_pgfault()** 在页面缺失（page fault）发生时进入该函数进行处理。
- **find_vma()** 查找虚拟地址addr是否在当前的虚拟进程中，返回值为null表示该地址不在进程有效的虚拟地址中。
- **get_pte()** 获取给定地址的PTE页表项，当页表项不存在时会分配新的页表。
- **pgdir_alloc_page** 为给定的地址分配物理页面并在此基础上将其映射到页表当中。
- **swap_in()** 从磁盘交换区加载数据，将数据加载到新的物理页面。
- **swapfs_read()** 从交换文件中读取数据（例如页面内容）并将其加载到内存中。在该函数中又调用**ide_read_secs()**用于从磁盘读取一定数量的扇区数据。
- **page_insert()** 将页面映射到对应进程的页表中，更新页表。
- **_fifo_map_swappable()** 设置页面为可交换状态,并将其插入到FIFO队列。
- **list_add()** 将page插到FIFO队列末尾，表示是最近被访问的页面。
- **_fifo_swap_out_victim()** 根据FIFO 算法选择最早进入内存的页面（队列头部的页面）作为被替换的页面进行交换（即替换出去）。
- **le2page()** 用于将 list_entry_t 转换为实际的 struct Page 指针。
- **swap_out()** 将页面进行换出，只有当没有空闲页时才进行。
- **swapfs_write()** 把要换出页面的内容保存到磁盘中，如果写入失败，则返回错误代码，并跳过该页面的处理。
- **tlb_invalidate()** 更新TLB。当页面被交换出去后，映射关系已经改变，因此需要使TLB中的该条目失效，避免后续对该页面的访问直接从TLB中查找。
- **assert()** 判断每一步执行过程是否正确，确保页面换入换出操作无误，否则进行断言中断。

2.练习2：深入理解不同分页模式的工作原理（思考题）

（1） 问题一：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？

a. sv32，sv39，sv48的异同?

相同点

- **基本架构**：SV32、SV39、SV48 都是 RISC-V 指令集架构中定义的虚拟内存管理方案，它们都基于页式内存管理。
- **页大小**：这三种模式都使用 4KB 作为基础页大小。
- **多级页表**：它们都采用了多级页表结构来管理虚拟地址到物理地址的映射。

不同点

- **虚拟地址空间大小**：
  - **SV32**：提供 32 位虚拟地址空间，支持最大 4GB 的虚拟地址空间。
  - **SV39**：提供 39 位虚拟地址空间，支持最大 512GB 的虚拟地址空间。
  - **SV48**：提供 48 位虚拟地址空间，支持最大 256TB 的虚拟地址空间。
- **页表级数**：
  - **SV32**：使用两级页表结构。
  - **SV39**：使用三级页表结构。
  - **SV48**：使用四级页表结构。
- **物理地址空间大小**：
  - **SV32**：提供 34 位物理地址空间，支持最大 16GB 的物理地址空间。
  - **SV39**：提供 56 位物理地址空间，理论上支持最大 64ZB 的物理地址空间，但实际上由具体实现决定。
  - **SV48**：同样提供 56 位物理地址空间，支持最大 64ZB 的物理地址空间。
- **适用场景**：
  - **SV32**：适用于对虚拟地址空间要求不高的嵌入式系统。
  - **SV39**：适用于大多数通用计算场景，如服务器、桌面系统等。
  - **SV48**：适用于需要超大虚拟地址空间的高端服务器和数据库系统。

b.两段代码为什么如此相似?

在上述代码中，`get_pte` 函数的两部分代码非常相似，主要体现在页表项的获取和页面的分配上。其相似性可以归结为以下几个方面：

相似的逻辑

1. **获取页表项（PTE）**
   - 在两部分代码中，代码首先根据虚拟地址 `la` 计算出对应的页目录项 `pdep1` 和 `pdep0`。这两个部分的代码都包含了对 `pgdir` （页表）的访问，分别访问不同级别的页目录项。
2. **检查页表项的有效性**
   - 都使用 `if (!(*pdep1 & PTE_V))` 和 `if (!(*pdep0 & PTE_V))` 来检查页目录项是否有效。如果页目录项无效，则需要分配新的物理页面，并为该页目录项分配新的页表项。
3. **分配新的页面**
   - 如果页表项无效，代码会通过调用 `alloc_page()` 分配一个新的物理页面，然后使用 `memset` 清空该页面，确保新分配的页面为空。

4. **创建新的页表项**

    - 在两段代码中，都使用 `pte_create` 创建新的页表项，将新分配的物理页面映射到虚拟地址空间。

代码的相似性

```
// 第一部分代码：检查并分配页目录项 pdep1
pde_t *pdep1 = &pgdir[PDX1(la)];
if (!(*pdep1 & 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);
    *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
}

// 第二部分代码：检查并分配页目录项 pdep0
pde_t *pdep0 = &((pde_t *)KADDR(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);
    *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
}
```

解释

- 相似的目标：两段代码的目的是相同的：检查页表项是否有效，如果无效，则分配新的物理页面并将其映射到虚拟地址。
- 区别：
    - 第一段代码操作的是一级页目录项 (pdep1)，而第二段操作的是二级页目录项 (pdep0)。
    - 它们分别在不同层级的页表中进行映射，确保虚拟地址空间的有效性。

（2） 问题二：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？

没有必要将两个功能拆开，理由如下：

- 两个功能合并在一起依旧保证了正确性，当查询页页表项存在时，直接返回，但是如果不存在时还会分配最新的页表项，然后再返回，这同时也实现了页表项的分配；

- 如果将两个功能分开，即将查询和分配分开，可能对于分配功能来说并没有什么实质性的改变，但是对于查询，如果我们仅仅写成如下代码：

```
pte_t *find_pte(pde_t *pgdir, uintptr_t la, bool create){
    /* blablabla */
pde_t *pdep1 = &pgdir[PDX1(la)];
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
    /* blablabla */
}
```

首先是不难发现，其中直接返回的页表项可能依旧是不存在的，因此会导致页表项的返回错误，或者仍需要调用页表项的分配算法先分配后再进行返回，因此还不如就保持将查询与分配放置在一个函数里面，这样即可以保证正确性，有可以减少不必要的调用。

3.练习3：给未被映射的地址映射上物理页（需要编程）

补充代码如下：

```
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.
            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            //(3) make the page swappable.
            swap_in(mm,addr,&page);
            page_insert(mm->pgdir,page,addr,perm);
            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;
        }
    }
```

通过其中的**get_pte**函数获取页表项，判断是否页表项为空项，如果为空则做下述的处理：

- 使用swap_in函数来将需要的物理页读入内存(当然如果内存不足，自动调用swap_out函数换出，最终一定能成功将该物理页换入)，然后使用page_insert来建立页表项到页之间的映射，最后把其可交换属性设置为真，插入FIFO的队列中。

（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。

- 选择替换的页：页替换算法需要选择一个页进行替换。存在位、访问位和已修改位可以帮助算法决定哪个页最不活跃或替换代价最低。
- 更新页表项：当页被替换时，相应的页表项需要更新以反映新的状态，例如，清除存在位，设置已修改位（如果页被写回磁盘）。
- 维护替换策略的状态：例如，时钟算法会周期性地清除访问位，并使用它们来追踪页的使用情况。

（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？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？

- 如果出现了页访问异常，那么硬件将引发页访问异常的地址将被保存在 cr2 寄存器中，设置错误代码，然后触发 Page Fault 异常，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，其中这两个异常的处理都会进入do_pgdefault函数进行处理。
- 是的，`Page` 结构的实例（通常构成一个数组）与页表中的页目录项和页表项有直接的对应关系 。对应关系说明：
    - `ref` **字段**：表示页帧的引用计数。这与页表项中的存在位（valid bit）有关，但不是直接对应。当页表项指向一个页帧时，引用计数可能会增加，表示有虚拟地址映射到该页帧。
    - `flags` **字段**：这是一个标志数组，描述了页帧的状态。这些标志可能与页表项中的标志位有直接对应关系，例如：
        - 是否脏（dirty）
        - 是否可写（read/write）
        - 是否存在（present）
        - 是否已访问（accessed）
    - `visited` **字段**：可能用于实现页面替换算法，比如时钟算法，这与页表项中的访问位（accessed bit）有间接关系。
    - `property` **字段**：表示空闲块的数目，通常用于内存分配算法（如首次适配）。这与页表项没有直接关系，但它是内存管理的一部分。
    - `page_link` **字段**：用于将页帧链接到空闲列表或已分配列表中。这与页表项没有直接关系，但它帮助操作系统管理内存。

- `pra_page_link` **字段**：用于页面替换算法（如页替换算法）。这与页表项没有直接关系，但它是虚拟内存管理的一部分。
- `pra_vaddr` **字段**：用于页面替换算法，存储与页帧关联的虚拟地址。这与页表项中的虚拟地址有间接关系，因为页表项定义了虚拟地址到物理地址的映射。

4.练习4：补充完成Clock页替换算法（需要编程）

基于给出的框架以及Clock页替换算法，修改代码如下：

_clock_init_mm()函数

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: 2211349*/
     // 初始化pra_list_head为空链表
     // 初始化当前指针curr_ptr指向pra_list_head，表示当前页面替换位置为链表头
     // 将mm的私有成员指针指向pra_list_head，用于后续的页面替换算法操作
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     list_init(&pra_list_head);
     curr_ptr=&pra_list_head;
     mm->sm_priv=&pra_list_head;
     cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

该函数实现初始化：

- 初始化**pra_list_head**为空链表，它将用于存储所有的页面。
- 使用**curr_ptr**用于指示时钟算法中的扫描位置，初始时指向 pra_list_head。
- 将表示进程的内存管理结构的**mm**指向 pra_list_head，以便在替换时访问页面链表。

clock_map_swappable()函数

```
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: 2212126*/
    // link the most recent arrival page at the back of the pra_list_head qeueue.
    // 将页面page插入到页面链表pra_list_head的末尾
    // 将页面的visited标志置为1，表示该页面已被访问
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_add(head, entry);
```

```
    page->visited = 1;
    return 0;
}
```

该函数实现了对页面链表的更新：

- 每当一个页面被映射并且变为可交换时就将其添加到链表中。
- 页面插入位置为链表尾部，并将页面设置为已经被访问过。

clock_swap_out_victim()函数

```
static int
_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);
     /* 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
    while (1) {
        /*LAB3 EXERCISE 4: 2213603*/
        // 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 = list_prev(curr_ptr);
            continue;
        }
        struct Page* curr_page = le2page(curr_ptr,pra_page_link);
        if(curr_page->visited == 0){
            cprintf("curr_ptr %p\n", curr_ptr);
            curr_ptr = list_prev(curr_ptr);
            list_del(list_next(curr_ptr));
            *ptr_page = curr_page;
            return 0;
        }
        curr_page->visited = 0;
        curr_ptr = list_prev(curr_ptr);
    }
    return 0;
}
```

该函数是Clock页替换算法的实现主要部分：

- 遍历**pra_list_head**链表，找到**第一个未被访问的页面**，选择该页面作为要被替换的页面
- 如果页面已经被访问过，则将其**visited标志置为 0**，并继续扫描下一个页面。

- 一旦找到一个未被访问的页面，将其从链表中删除并将该页面的指针赋给**ptr_page**，表示该页面将被换出。
- 如果扫描回到了链表的头部，**curr_ptr重新开始扫描。**

5.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？

（1）优势

1. 减少页表项数量

- **减少内存占用**：由于每个大页覆盖了更大的地址空间，因此需要更少的页表项来映射整个虚拟地址空间，这样可以减少页表占用的内存。

2. 提高缓存效率

- **提高TLB命中率**：大页减少了页表项的数量，这意味着更多的页表项可以存储在转换后备缓冲器（TLB）中，从而提高TLB的命中率，减少地址转换的开销。

3. 简化内存管理

- **减少页表维护**：大页减少了页表项的数量，简化了内存管理，降低了页表维护的复杂性和开销。

4. 提升性能

- **减少页表遍历**：对于大页映射，CPU在遍历页表时需要进行的内存访问次数减少，这可以提升性能。

风险

1. 内部碎片

- **空间浪费**：由于大页的大小固定，如果进程没有完全使用一个大页的空间，就会产生内部碎片，导致内存浪费。

2. 灵活性降低

- **映射灵活性**：大页映射方式不如分级页表灵活，对于需要精细映射的场合，大页可能不适用。

3. 分配策略复杂

- **分配策略**：大页的分配策略可能比普通页复杂，因为需要考虑大页的固定大小和内存的连续性要求。

4. 性能问题

- **页替换效率**：如果一个大页中的某些部分频繁被访问而其他部分很少被访问，那么整个大页都需要保留在内存中，这可能导致页替换算法效率低下。

5. 安全性问题

- **地址空间隔离**：大页可能导致地址空间隔离粒度降低，如果一个进程能够利用大页中的漏洞访问到不应该访问的内存区域，可能会引发安全问题。

总结来说，大页页表映射方式在减少内存占用和提高缓存效率方面具有明显优势，但也存在内部碎片、灵活性降低和安全性问题等风险。

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）

我们添加了**swap_lru.h**和**swap_lru.c**来实现LRU页替换算法。基于FIFO算法进行改编，在page加入链表中时LRU算法：

- 每当页面被访问时，我们需要将其移动到链表的头部（表示最新访问）。
- 如果页面已在链表中，我们需要**删除原来的节点**，然后**重新插入**到链表头部 主要的代码如下：

```
#include <defs.h>
#include <riscv.h>
#include <stdio.h>
#include <string.h>
#include <swap.h>
#include <swap_clock.h>
#include <memlayout.h>
#include <list.h>
/* LRU (最近最少使用) 页面替换算法 */
list_entry_t pra_list_head, *curr_ptr;
/*
 * (1) _lru_init_mm: 初始化LRU页面替换结构
 */
static int
_lru_init_mm(struct mm_struct *mm) {
    list_init(&pra_list_head);
    curr_ptr = &pra_list_head;
    mm->sm_priv = &pra_list_head;
    cprintf(" mm->sm_priv %x in lru_init_mm\n", mm->sm_priv);
    return 0;
}
/*
 * (2) _lru_map_swappable: 将最近访问的页面添加到LRU队列的前端
 */
static int
_lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in) {
    list_entry_t *entry = &(page->pra_page_link);
    assert(entry != NULL && curr_ptr != NULL);

    list_entry_t *head = (list_entry_t*) mm->sm_priv;
    list_entry_t *temp = head->next;
    while(temp != head) //
    {
        if(entry == temp)
        {
            list_del(temp);
            break;
        }
        temp = temp->next;
    }
```

```
    list_add(head, entry);  // 将页面插入到链表的前端
    return 0;
}
/*
 * (3) _lru_swap_out_victim: 选择最久未访问的页面进行替换
 */
static int
_lru_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);

    // 遍历LRU链表，找到最久未被访问的页面（位于链表尾部）
    list_entry_t *victim_entry = list_prev(head);


    struct Page *victim_page = le2page(victim_entry, pra_page_link);
    list_del(victim_entry);
    *ptr_page = victim_page;

    return 0;
}
```