Lab3实验报告

练习1：理解基于FIFO的页面替换算法

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？

`do_pgfault()` 函数，是页面置换机制的核心。如果过程可行，没有错误值返回，我们就可对页表做对应的修改，通过加入对应的页表项，并把硬盘上的数据换进内存，这时还涉及到要把内存里的一个页换出去，而 `do_pgfault()` 函数就实现了这些功能。

换入换出流程：

1、当发生缺页异常时，进入 `trap.c` 文件执行，出现异常的地址传入 `do_pgfault()` 函数进行缺页处理，使用 `get_pte` 函数，获取或创建给定虚拟地址对应的页表项。

2、如果页表项为空，调用 `pgdir_alloc_page` 函数分配物理页面并设置映射。 `pgdir_alloc_page` 函数内调用 `alloc_page()` 函数来分配一个新的物理页面，并将返回的页面结构体指针存储在 `page` 变量中。 `alloc_page()` 函数中无限循环，直到成功分配页面或满足某些退出条件，当页面数量不够时，调用 `swap_out` 函数，将一些页面从内存中换出到硬盘，以便腾出更多的物理内存供后续分配使用。 `swap_out` 函数中调用 `swap_out_victim` 函数，从内存中选择一个合适的页面作为换出页面，并且调用 `swapfs_write` 函数将页面数据写入硬盘。

如果成功分配到一个页面，则调用 `page_insert()` 函数将新分配的页面插入到指定的页目录中，映射到给定的线性地址，并设置权限。如果页面插入失败，释放已经分配的页面。交换空间进行初始化，调用 `swap_map_swappable()` 函数将页面映射到交换空间中，返回分配并映射成功的页面结构体指针。

3、如果待处理的页表项已存在，意味着该地址对应的页在之前被换出，此时调用 `swap_in` 函数从磁盘加载数据到内存页，同时调用 `page_insert` 函数建立物理地址与逻辑地址的映射， `swap_map_swappable` 函数设置页面可交换。

FIFO页面置换算法换入换出流程中各个函数的作用

1、 `do_pgfault`： 发生 `page fault` 后，异常处理器会调用 `do_pgfault` 函数，是页面置换机制的核心，对页表进行修改，换入换出页，都是在这个函数中进行，以下函数均为 `do_pgfault` 函数中使用到的关键函数。

2、 `find_vma`： 用于在内存管理结构 `mm_struct` 中查找包含指定虚拟地址 `addr` 的虚拟内存区域VMA。

3、 `get_pte`： 用于获取指定虚拟地址la对应的页表项，如果页表项为空，则分配一个新的页并储存映射关系。

4、 `pgdir_alloc_page`： 用于分配和初始化页面，其中调用 `alloc_page` 函数。

5、 `alloc_page`： `alloc_page` 为 `alloc_pages` 的一个宏，该函数为给定的虚拟地址分配一个新的物理页面。如果页面数量不够，则会调用 `swap_out` 函数换出所需的页面数量。

6、 `swap_in`： 用于根据给定的虚拟地址 `addr` 找到对应交换条目，从磁盘读取数据到一个新分配的物理页面，并返回该物理页面指针。

7、 `page_insert`： 基于新分配的页面，更新页表项建立虚拟地址到物理地址的映射。

8、 `swap_map_swappable`： FIFO算法中直接调用 `_fifo_map_swappable`，将一个物理页面标记为可交换，并将这个新页面存入队列，队列保证FIFO算法先进先出原则，以便在需要时进行页面交换。

9、`swap_out`：用于将指定数量的物理页面的数据写入磁盘，并释放这些物理页面内存空间。

10、`swapfs_read`：用于从磁盘中读取页面数据到内存，调用`ide_read_secs`函数，该函数是IDE设备读取函数，用于从磁盘读取扇区数据。

11、`swapfs_write`：用于将页面数据写入磁盘，调用`ide_write_secs`函数，该函数是IDE设备写入函数，用于将数据写入磁盘扇区。

12、`swap_out_victim`：FIFO算法中直接调用`_fifo_swap_out_victim`，用于从FIFO队列中选择一个最早的页面作为换出的页面，并返回该页面指针。

13、`tlb_invalidate`：TLB是一个高速缓存，用于加速地址映射过程。当页表项发生变化时，需要使TLB中的相应条目失效，以确保后续的地址翻译使用最新的页表项。

14、`free_page`：`free_page`为`free_pages`的一个宏，用于将指定的物理页面释放到物理内存管理器中，以便这些页面可以被重新分配和使用。

练习2：深入理解不同分页模式的工作原理

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。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
pde_t *pdep1 = &pgdir[PDX1(la)];//计算虚拟地址 la 对应的页目录项（PDE）的索引，并取得该项的地址
    if (!(*pdep1 & PTE_V)) { //检查页目录项是否有效
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        //如果目录项不存在且 create 为真，尝试分配一个新的物理页。如果分配失败，返回 NULL。
        set_page_ref(page, 1); //页面引用计数为 1
        uintptr_t pa = page2pa(page); //初始化物理页内存
        memset(KADDR(pa), 0, PGSIZE);  //更新页目录项
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);  //设置其为有效且可用户访问
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//获取页表项（PTE）
//    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)];
}
```

首先讨论sv32，sv39，sv48的异同，sv32使用两级页表，包括页目录和页表，虚拟地址分为页目录索引（10位）和页表索引（10位），加上页内偏移（12位）两部分；SV39使用三级页表包括页目录指针表、页目录和页表，而虚拟地址分为三部分：页目录指针表索引（9位）、页目录索引（9位）、页表索引（9位），加上页内偏移（12位）；SV48使用四级页表（页映射级别4表、页目录指针表、页目录和页表）虚拟地址分为页映射级别4表索引（9位）、页目录指针表索引（9位）、页目录索引（9位）、页表索引（9位），加上页内偏移（12位）四部分。

而`get_pte()`函数主要执行的是根据给定的虚拟地址（`la`）和是否创建新页表项（`create`）的标志，找到或创建对应的页表项（PTE）。虽然不同的页表模式在页表层级上有所不同，但由于分页机制就是通过多级页表逐步将虚拟地址映射到物理地址，而每一级页表的处理逻辑是相似的，因此作为执行分页机制中每一级页表的处理函数，可以设计的较为相似，从而能够适应不同级别的页表。

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？

将页表项的查找和分配合并在一个函数`get_pte()`中，可以简化接口，减少了函数调用次数，同时可以减少重复代码，提高代码重用性的同时增强了代码的可读性。

但拆分成独立的函数可以使得这两个逻辑上相对独立的操作能更加灵活的使用，应用场景更加广泛，也可以在发生错误时更好的进行测试和处理。

练习3：给未被映射的地址映射上物理页

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。

简要说明你的设计实现过程

在kern/mm/vmm.c中填写代码如下：

```
if (swap_in(mm, addr, &page) != 0 ){
    cprintf("swap_in in do_pgfault failed\n");
    goto failed;
}
// 更新页表项，建立映射关系
if (page_insert(mm->pgdir, page, addr, perm) != 0){
     cprintf("page_insert in do_pgfault failed\n");
     goto failed;
}
// 设置页面可交换
swap_map_swappable(mm, addr, page, 1);

page->pra_vaddr = addr;
```

这一部分代码是在进入`do_pgfault`函数处理缺页异常情况后，找到虚拟地址对应的`vma`且存在一个页表项的情况，过程如下：

- 首先需要将一个磁盘页换入到内存中，调用`swap_in`函数，根据页表基地址和虚拟地址从磁盘对应位置读取，写入内存。

- 建立页表项的映射关系，`page_insert`函数将虚拟地址与内存中的物理页进行映射，更新页表项，并刷新TLB。
- 最后调用`swap_map_swappable`，设置页面可交换。

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

在页替换算法的过程中，主要依赖与操作的是mm结构体，该结构体把一个页表对应的信息组合起来。而对于PDE和PTE，可以看到在kern/mm/mmu.h中有一些宏定义。

```
// page directory index
#define PDX1(la) ((((uintptr_t)(la)) >> PDX1SHIFT) & 0x1FF)
#define PDX0(la) ((((uintptr_t)(la)) >> PDX0SHIFT) & 0x1FF)

// page table index
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x1FF)

// page table entry (PTE) fields
#define PTE_V     0x001 // Valid
#define PTE_R     0x002 // Read
#define PTE_W     0x004 // Write
#define PTE_X     0x008 // Execute
#define PTE_U     0x010 // User
#define PTE_G     0x020 // Global
#define PTE_A     0x040 // Accessed
#define PTE_D     0x080 // Dirty
#define PTE_SOFT  0x300 // Reserved for Software
```

这些宏定义是根据虚拟地址进行位移，找到其对应的页表与页目录表的下标位置，以及标志位的定义。

在函数`gte_pte`中可以看到对宏定义的应用。根据虚拟地址，利用宏定义，在三级页表中逐级向下查找，最终找到其对应的页表项。

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

此外，PDE与PTE中的标识位也为算法提供了判断条件，如`PTE_V`可用于检查该页表项是否有效。在`get_pte`中，对无效的页表项会创建一个新的页面。

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?

- 保存当前CPU状态，切换到内核态准备处理异常，将异常相关信息打包成`tf`结构体。
- 根据`stvec`寄存器中的地址跳转到中断处理程序，也就是`trap.c`文件中的`trap`函数。
- 根据`tf->cause`字段判断是中断还是异常，这里是异常，会进入`exception_handler`进行处理。
- 再根据`cause`字段判断异常的具体类型，是一个缺页异常，在相应的`case`中调用`pgfault_handler`，再到`do_pgfault`进行详细的缺页异常处理。
- 处理结束后，成功则回到异常位置继续执行，否则会输出报错信息`unhandled page fault`。

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?

在sv39分页机制下，每一个页表刚好是一个4kB的页，而在`get_pte`中我们也看到，如果按偏移量找到的位置页表项无效，就会为其分配一个页。在本次实验中，每一个页表项或页目录项，无论它指向下一级的页表还是一个物理页，实际上都是Page数组的其中一项。

练习4：补充完成Clock页替换算法

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)

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: 2211849*/
     // 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;
     return 0;
}
/*
 * (3)_fifo_map_swappable: According FIFO PRA, we should link the most recent
arrival page at the back of pra_list_head qeueue
 */
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);
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    assert(entry != NULL && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: 2211849*/
    // 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_add(head, entry);
    page->visited = 1;
    return 0;
}
/*
 *  (4)_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
_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: 2211849*/
        // 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
        list_entry_t *entry = head->prev;

        if (entry == head) {
            *ptr_page = NULL;
            break;
        }
        struct Page* page = le2page(entry,pra_page_link);// 获取页面指针
        if(!page->visited)
        {
            list_del(entry);
            *ptr_page = page;
            cprintf("curr_ptr %p\n", curr_ptr);
            break;
        }
        if (page->visited)
        {
            page->visited =0;
            curr_ptr = entry;

        }
```

```
        entry = entry->prev;
    }
    return 0;
}
```

设计实现过程

1.初始化页面链表，存储可替换的所有页面，并且维护一个curr_ptr指针记录当前页面替换的位置。同时创建一个循环链表，用于保存物理页框。

2.映射可替换页面，当页面被加载到内存中时，将其增加到页面链表的末尾，同时将visited置为1，表示被访问。

3.当页面换出时，遍历链表，检查当前指向页面的访问位，如果找到一个未被访问的页面，则将其从链表中删除，并将该页面的指针赋值给 `ptr_page`，作为换出的页面。如果页面已被访问，则将其 `visited` 标志重置为 0，表示这次检查已经"访问"过这个页面，并将 `curr_ptr` 更新为当前页面的位置，以便下次从这里开始检查。如果遍历了整个链表而没有找到未被访问的页面，则 `ptr_page` 被设置为 `NULL`，表示没有找到合适的页面进行替换。

比较Clock页替换算法和FIFO算法的不同

1. clock算法实现较为复杂，维护了一个缓冲区来模拟时钟，页面在缓冲区中按访问顺序排列。FIFO算法只使用了一个链表来维护页面队列。
2. clock算法考虑了页面的访问顺序，可以保留最近被访问的页面，而fifo算法只考虑了页面进入的顺序。
3. fifo算法实现可能会发生`Belady's Anomaly`，而clock算法可以尽量避免该现象的发生。

练习5：阅读代码和实验手册，理解页表映射方式相关知识

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？

优势：

- **简化查找过程**：由于减少了页表层级的访问，大页映射方式可以降低页表查找的次数，从而减少内存访问的开销，缩短页表的访问时间。
- **实现简单：**大页映射的实现更加简单，不需要复杂的多级页表管理算法，操作系统只需维护一个大型页表即可。这降低了复杂性，也简化了地址转换的实现。
- **减少TLB失效开销**：减少了多级页表中的页表项数量，相当于每个页面的映射关系直接存在一个大表中，因此，TLB缓存未命中时只需要进行一次页表查询，减少了每次TLB失效时多级页表的开销，从而提高了TLB利用率。

劣势：

- **内存开销较大**：每次分配页面需要分配出去的内存较大，对系统的内存需求就大，会导致巨大的内存开销。
- **内存空间浪费：**为程序分配一个页时，程序可能不会使用到全部的虚拟空间，这就会导致一部分内存被浪费掉，对于一个大页，这种浪费更加严重。
- **灵活性差**：分级页表可以灵活地映射不同大小的内存块，来适应不同大小的内存需求，而大页映射限制了这种灵活性。

扩展练习 Challenge：实现不考虑实现开销和效率的LRU页替换算法

LRU算法原理

LRU算法核心思想在于当内存空间不足时，系统会淘汰那些最近最少被使用的页面，以此释放内存空间供新的页面使用。这种算法的基础在于假设了一个前提：如果一个数据项在最近一段时间内未被访问到，那么在将来它被访问的可能性也很小。

LRU算法使用的数据结构

使用双向链表，链表的两端分别是最近使用和最久未使用的页面。当某页面被访问时，将该页面移动到链表的头部。当需要进行页面置换时，选择链表尾部的页面进行置换。

LRU算法实现思路

```
// 将页面移动到链表头部
static void _lru_update_page(struct mm_struct *mm, struct Page *page) {
    list_entry_t *head = (list_entry_t *)mm->sm_priv;
    list_entry_t *entry = &(page->pra_page_link);

    assert(entry != NULL && head != NULL);

    // 遍历链表，检查是否存在指定的页面
    list_entry_t *curr;
    for (curr = head->next; curr != head; curr = curr->next) {
        struct Page *curr_page = le2page(curr, pra_page_link);
        if (curr_page == page) {
            break;  // 找到页面，退出循环
        }
    }

    // 如果页面不存在于链表中，退出函数
    if (curr == head) {
        return;
    }

    // 如果页面已经在链表头部，不需要移动
    if (entry->prev == head) {
        return;
    }

    // 从链表中删除页面
    list_del(entry);

    // 将页面插入到链表头部
    list_add(head, entry);
    return 0;
}
```

增加一个函数，在访问一个页面时循环遍历链表各节点判断该页是否在内存中，如果存在，将该页面移至链表头部，表示最近被访问。若不存在，则退出该函数，执行将链表尾部页面删除操作，将该页加入到链表头部。根据以上算法可实现发生缺页异常时，将最久未使用的页面淘汰，以此释放内存空间供新的页面使用。

经过多次尝试，我们发现此次实验无法实现页面的访问，因为本次实验所有的操作都是基于缺页故障进行的，仅实现了简单的页面置换机制，并没有涉及内核线程和用户进程，所以无法进行当前进程的页面访问，故无法调用该访问函数，但经过思考，在整个用户态的页面访问实现过程中，该LRU算法思路是可以实现的。