操作系统Lab3

练习0：填写已有实验

经过仔细代码查阅，发现这里实际上并不需要补充lab2中的代码到这里，其中对lab3造成影响的代码已经均在lab3的基础代码中，所以这里不再需要对lab2的代码进行转移。

页表项的结构在指导书里说明的很充分，这里不再赘述。

练习1：理解基于FIFO的页面替换算法

换入换出流程

简述整个虚拟内存管理的过程：当发生缺页异常时，出现异常的地址会传入 `do_pgfault` 函数进行缺页的处理。此时使用 `get_pte` 函数获取/创建对应虚拟地址映射过程中的页表和页表项：

1. 若得到的页表项为 0，表示这个页从来没有被分配，此时使用 `pgdir_alloc_page` 分配物理页并且使用 `page_insert` 建立映射。此外注意到在 `alloc_pages` 中增加了一个循环，当页面数量不够时使用 `swap_out` 函数将对应数量的页面换出。在 `swap_out` 中会调用页面置换管理器的 `swap_out_victim` 按照对应的策略换出页面，并且使用 `swapfs_write` 将页面内容写入“硬盘”中。

2. 若得到的页表项不为 0，表示这个地址对应的页是在之前被换出的（关于页表项值的操作见后文对 `swap_out` 具体操作的分析）。此时需要调用 `swap_in` 将页面内容从“硬盘”中通过 `swapfs_read` 读入对应的页面，并且使用 `page_insert` 将虚拟地址和物理页面的映射关系写入对应的页表项中。之后再使用 `swap_map_swappable` 调用不同策略的 `map_swappable` 进行维护，从而保证页面置换能够正确执行。

总的来说，在执行到 `*addr=...` 的时候，首先触发缺页异常，进入trap.c依次执行并调用核心处理函数 `do_pgfault` ，然后获取vma和mm等管理结构，通过 `get_pte` 获取或创建addr对应的物理页和页表项，如果原本已经有页表项，那么就把原来的页换到磁盘上去，新的这个页换进内存，结束。

描述每个函数在过程中做了什么

在上文，我分析了虚拟内存管理的流程。但是具体的页面置换算法会在下面的swap_fifo中具体进行实现。具体实现的方法可以在 `swap_manager` 内的看到，你可以用fifo对应的实现或者clock对应的实现来赋值 `swap_manager` 。现在我以FIFO算法为例具体地依次分析 FIFO 页面替换算法中页面换入到被换出过程中的十个函数（由于要求使用简单的一两句话描述，完整的描述放在附录中）：

1. `do_pgfault`：整个缺页处理流程的开始，根据 `get_pte` 得到的页表项的内容确定页面是需要创建还是需要换入。
2. `find_vma`：find_vma函数找到包含传入的错误地址的VMA，还会更新。
3. `get_pte`：根据触发缺页异常的虚拟地址查找其对应的多集页表叶节点的页表项。如果其中某一级页表不存在则为其分配一个新的页（4KiB）用于存储映射关系。
4. `swap_in`：将对应的页面根据存储在页表项位置的 `swap_entry_t` 9从“硬盘”中读取页面内容，并且重新写入页面的内存区域。
5. `swapfs_write`：其会调用 `ide_write_secs` 函数， `ide_write_secs` 的主要目标是模拟向IDE设备（如硬盘）的特定扇区写入数据。
6. `alloc_page` / `alloc_pages`：前者为后者的一个宏，会分配一个页面，如果不能分配则使用 `swap_out` 换出所需的页面数量。
7. `page_insert`：将虚拟地址和新分配的页面的物理地址在页表内建立一个映射。
8. `swap_map_swappable`：在使用 FIFO 页面替换算法时会直接调用 `_fifo_map_swappable` ，这会将新加入的页面存入 FIFO 算法所需要维护的队列（使用链表实现）的开头从而保证先进先出的实现。
9. `swap_out`：根据需要换出的页面数量换出页面到“硬盘”中。

10. `swapfs_read`：其会调用`ide_write_secs`函数，`ide_write_secs`的主要目标是模拟从IDE设备（如硬盘）的特定扇区读入数据。
11. `sm->swap_out_victim()`：在 FIFO 中会直接调用`_fifo_swap_out_victim`进行页面换出，这会将链表最后（最先进入的页面）指定为需要被换出的页面。
12. `free_page`：将被换出的页面对应的物理页释放从而重新尝试分配。
13. `tlb_invalidate`：在页面换出之后刷新 TLB，防止地址映射和访问发生错误。

触发page_fault的部位：

```
static inline void
check_content_set(void)
{
     //将数字0x0a写入虚拟地址0x1000
     *(unsigned char *)0x1000 = 0x0a;
     assert(pgfault_num==1);
     *(unsigned char *)0x1010 = 0x0a;
     assert(pgfault_num==1);
     *(unsigned char *)0x2000 = 0x0b;
     assert(pgfault_num==2);
     *(unsigned char *)0x2010 = 0x0b;
     assert(pgfault_num==2);
     *(unsigned char *)0x3000 = 0x0c;
     assert(pgfault_num==3);
     *(unsigned char *)0x3010 = 0x0c;
     assert(pgfault_num==3);
     *(unsigned char *)0x4000 = 0x0d;
     assert(pgfault_num==4);
     *(unsigned char *)0x4010 = 0x0d;
     assert(pgfault_num==4);
}
```

练习2：深入理解不同分页模式的工作原理

解释两段代码为什么如此相像

RISC-V 中目前共有四种分页模式 Sv32，Sv39，Sv48，Sv57。其中 Sv32 使用了二级页表，Sv39 使用了三级页表，Sv48 和 Sv57 则是使用了四级和五级页表。在我们的ucore实验中使用的是**Sv39模式**，页表被分为了：PDX1、PDX0和PTX。

`get_pte` 的目的是查找/创建特定线性地址对应的页表项。由于 ucore 使用 Sv39 的分页机制，所以共使用了三级页表，`get_pte` 中的两段相似的代码分别对应于在第三、第二级页表（PDX1，PDX0）的查找/创建过程：

- 首先 `get_pte` 函数接收一个参数 `create` 标识未找到时是否要分配新页表项。之后首先通过 `PDX1` 宏获取线性地址 `la` 对应的第一级 `PDX1` 的值，并且在 `pgdir` 即根页表（页目录）中找到对应的地址：

```
1 // 找到对应的Giga Page
2 pde_t *pdep1 = &pgdir[PDX1(la)];
```

  并且判断其 `V` 标志位是否为真（表示可用）。若可用则继续寻找下一级目录，若不可用则根据 `create` 的值决定是否调用 `alloc_page()` 开辟一个新的下一级页表，之后设置这一级页表的页表项，页面的引用以及下一级页表的页号的对应关系。
- 之后再对下一级页表进行同样的操作，其中通过 `PDE_ADDR` 得到页表项对应的物理地址。
- 最后根据得到的 `pdep0` 的页表项找到最低一级页表项的内容并且返回。

因此可以看出，这两段代码如此相像的原因为：由于sv39为三级页表，get_pte函数的目标是从虚拟地址找到pte的地址，所以**首先会使用pgdir（指向PDX1）找到PDX0的地址，然后再通过PDX0找到PTX的地址，由于页表的索引特性，这就导致了这两段代码的**

结构非常相似。最后通过PTX(la)索引就得到了pte的地址。

整个 `get_pte` 会对 Sv39 中的高两级页目录进行查找/创建以及初始化的操作，并且返回对应的最低一级页表项的内容。两段相似的代码分别对应了对不同级别 PDX（叫PDE也行）或 PTX （叫PTE也行）的操作。

查找和分配的功能

经过探讨，我们小组认为这种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的写法**并不好**。虽然对于ucore中需要实现的sv39分页机制，这种写法能够实现功能且能够增强代码的复用，减少可能出现的错误，但是可扩展性较差，例如分配可能失败，但函数需要返回页表项指针，这会带来隐含的假设分配一定成功，代码难以维护。有两种更加细粒度的函数可以考虑实现：

- 所以我认为应该将这两个功能拆开为两个单独的函数如：`pte_lookup(mm, addr)` 仅用于查找给定地址的页表项，如果不存在则返回NULL。`pte_alloc(mm)` 专门用于分配新的页表项，失败需要返回错误码。
- 可以将创建某一级页表的代码独立成一个模块，在查找中调用，这样会让代码的可扩展性提高。

练习3：给未被映射的地址映射上物理页

在kern/mm/vmm.c文件中，代码实现如下：

```
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.
    //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);

    //(2) According to the mm,addr AND page, setup the map of phy addr <--->logical addr
    page_insert(mm->pgdir, page, addr, perm);
    //更新页表，插入新的页表项

    //(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;
}
```

设计思路：

`do_pgfault` 函数处理缺页异常情况，其在出现异常根据 `scause` 寄存器中的两个分类 `CAUSE_LOAD_PAGE_FAULT` 和 `CAUSE_STORE_PAGE_FAULT` 时被 `exception_handler` 下的 `pgfault_handler` 调用，用来尝试进行页面替换。

设计实现过程：

1. 首先我们需要将一个磁盘页调到内存中，调用 `swap_in()` 函数 ，用 mm_struct *mm 和 addr 作为参数调用 `get_pte` 查找/构建该虚拟地址对应的页表项，并通过 `alloc_page()` 分配物理页和 `swapfs_read()` （本质上是memcpy）将数据从硬盘读到内存页result中，&page利用传参完成赋值，保存换入的物理页面。
2. 我们需要将换入的物理页添加虚拟页映射，调用 `page_insert` 函数更新页表/插入新的页表项并刷新TLB。在 `page_insert` 中还是先找该虚拟地址对应的页表项，如果valid（说明这地方有个PTE）则检查该页是否和要插入的页相同；最后调用 `*ptep = pte_create(page2ppn(page), PTE_V | perm)` 建立该页对应的PTE。
3. 设置该页面可交换：找到swap.c中的 `swap_map_swappable` 函数，设置好对应参数即可调用。

对以下三个问题的解答：

- Q1: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。

我们可以看到，实现页替换算法函数的关键参数主要是struct mm_struct *mm，该结构体把一个页表对应的信息组合起来，包括按虚拟地址大小排序的线性表mmap_list，`vma_struct`链表的首指针mmap_cache，对应的页表在内存里的指针pgdir，`vma_struct`链表的元素个数map_count和给swap manager应用的私有数据void *sm_priv。

通过 struct mm_struct *mm 和39位虚拟地址 addr ，我们能调用`get_pte()`查找某个虚拟地址对应的页表项，如果不存在这个页表项，会为它分配各级的页表；同理，swap.c中的swap_in，swap_out函数的重要参数之一都是mm；实现clock页替换算法文件 swap_clock.c 中的初始函数、页面插入和排除函数也要用到mm->sm_priv。

PDE（因为riscv是三级页表，页目录项应该叫做PDX1和PDX0）和PTE可以很快的加速页替换算法中的诸多判断，如下宏定义和相关判断：

```
// 计算PDX1,0的下标
#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) 相关字段值
#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
```

以上通过对虚拟地址la的位移和与操作进行页目录项和页表项下标的计算。

在调用`get_pte`查找，分配页表项的时候对PDX1,PDX0和PTE做的判断最多：

```
//get_pte查找某个虚拟地址对应的页表项，如果不存在这个页表项，会为它分配各级的页表
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {

    pde_t *pdep1 = &pgdir[PDX1(la)];//找到对应的Giga Page
    if (!(*pdep1 & PTE_V)) {
    //如果下一级页表不存在，那就给它分配一页，创造新页表
        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);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
        //注意这里R,W,X全零
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
    //再下一级页表（更多可能是页目录）
    if (!(*pdep0 & PTE_V)) {
    //思路同上，建立页表
        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);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

可以看到通过我们预先的宏定义加速了判断，并使得代码逻辑清晰。

并能通过pgdir_alloc_page快速给对应的PTE分配Page：

```
struct Page *pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page();
    if (page != NULL) {
        if (page_insert(pgdir, page, la, perm) != 0) {
            free_page(page);
            return NULL;
        }
        if (swap_init_ok) {
            swap_map_swappable(check_mm_struct, la, page, 0);
            page->pra_vaddr = la;
            assert(page_ref(page) == 1);
        }
    }

    return page;
}
```

通过调用 alloc_page 和 page_insert 两个函数用参数 PDE 分配对应的页面，更高一层的抽象带来更好的简洁性。

- Q2: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?

缺页异常是指CPU访问的虚拟地址时， MMU没有办法找到对应的物理地址映射关系，或者与该物理页的访问权不一致而发生的异常。在 kern_init 中初始化物理内存后调用 idt_init 初始化中断描述表，将寄存器 sscratch 赋值为0表示现在执行的是内核代码；将异常向量地址 stvec 设置为 &__alltraps 并将 sstatus 设置为内核可访问用户内存。

出现页访问异常时，trapframe 结构体中的相关寄存器被修改，按照一般异常处理的流程（见lab1，中断入口点），产生异常的指令地址会存入 `sepc` 寄存器，访问的地址存入 `stval`，并且根据设置的 `stvec` 进入操作系统的异常处理过程。于是 `kern/trap/trap.c` 的 `exception_handler()` 被调用，根据 trapframe *tf->cause（也就是scause）的值 CAUSE_LOAD_PAGE_FAULT /CAUSE_STORE_PAGE_FAULT调用 pgfault_handler 进行进一步处理。

- Q3:数据结构Page与页表的关系：

我们在物理页面管理中使用了 default_init_memmap 向内存里分配了一堆连续的 Page 结构体，来管理物理页面。可以把它们看作一个结构体数组。extern struct Page *pages 指针是这个数组的起始地址。

因此，使用下面的 page2ppn 可以运算得到正确的物理页号，左移若干位就可以从物理页号得到页面的起始物理地址。

```
// page2ppn
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }

// page2pa
static inline uintptr_t page2pa(struct Page *page)
    {return page2ppn(page) << PGSHIFT; }
```

Page结构体的定义在kern/mm/memlayout.h中

```
struct Page {
    int ref;                        // page frame's reference counter
    uint_t flags;                 // array of flags that describe the status of the page frame
    uint_t visited;                 //为clock新增
    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)
};
```

其中的 ref 是页帧引用次数记录，flags是页帧的使用状态；

sv39 分页机制下，每一个页表所占用的空间刚好为一个页的大小。在处理缺页时，如果一个虚拟地址对应的二级、三级页表项（页目录项）不存在，则会为其分配一个页，当第一级页表项没有设置过时也会分配一个页（主要体现在上文的 `get_pte` 中）。此外，当一个页面被换出时，他所对应的页面会被释放，当一个页面被换入或者新建时，会分配一个页面。所以，对于本实验中缺页机制所处理和分配的所有页目录项、页表项，都对应于 `pages` 数组中的一个页，但是 `pages` 中的页并不一定会全部使用。

练习4：补充完成Clock页替换算法

主要完成kern/mm/swap_clock.c中的 `_clock_init_mm`，`_clock_map_swappable` 和 `_clock_swap_out_victim` 三个函数，关键实现如下：

- `_clock_init_mm`：

```
static int
_clock_init_mm(struct mm_struct *mm)
{
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // 初始化pra_list_head为空链表
    list_init(&pra_list_head);
    // 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
    curr_ptr=&pra_list_head;
    // 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
    mm->sm_priv = &pra_list_head;
    //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
    return 0;
}
```

- `_clock_map_swappable`：

```
static int
_clock_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 && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // 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

    //list_add(head, entry);
    list_add(head -> prev, entry);
    //list_add_before(head, entry);

    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited=1;
    return 0;
}
```

- `_clock_swap_out_victim`：

```c
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: 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
        if (curr_ptr == &pra_list_head) {
            curr_ptr = list_next(curr_ptr);
        }
        // 获取当前页面对应的Page结构指针
        struct Page *ptr = 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
        // 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(ptr->visited){
            ptr->visited=0;
        }else{
            cprintf("curr_ptr %p\n",curr_ptr);//按照指导书要求，输出curr_ptr的值
            //list_del(curr_ptr); //不应该直接删除，因为下一次处理的时候没有移动到下一个节点
            curr_ptr = list_next(curr_ptr);
            list_del(curr_ptr->prev);
            *ptr_page=ptr;
            break;//这里break后就不执行后面的list_next，不会多走
        }
        curr_ptr = list_next(curr_ptr);
    }
    return 0;
}
```

下面介绍**Clock 页替换算法的实现思路 和 比较Clock页替换算法和FIFO算法的不同：**

对于页面链表 pra_list_head 来说，clock页替换算法在每次加入新页时添加到链表的末尾，而执行换出页面时从**全局变量**当前指针 curr_ptr 开始查找最早未被访问的页面。则 _clock_swap_out_victim 函数中的循环意图即为从当前指针开始查找，如果该页面已被访问则将 visited 状态设置为0起到重置作用，如果该页面未被访问，则调用 list_del 函数将该页面从页面链表中删除，并将该页面指针赋值给参数 ptr_page（*ptr_page=ptr）作为换出页面。这样即使链表中所有页之前都被访问过，也会按照该逻辑遍历一圈后删除 curr_ptr 指向的页面。

与之相反的是FIFO算法，其每次添加新页时添加到链表的头部，每次换出页面时调用 list_prev 找链表头的**前一个**节点（因为是双向链表，直接找到链表尾）来进行换出，如果该页面不是链表头（是链表头意味着只有一个元素，按照换出逻辑来说不存在该情况）则将该页面赋值给参数ptr_page作为换出页面；

注意到 curr_ptr 的本质作用也是指向最老的页面，不过其会随着链表的插入删除变化，而不是每次都从链表头遍历。当然更关键的是clock考虑了**页面的访问情况**，而不是像FIFO一样粗暴的驱逐最早进入的页面。

练习5

采用"一个大页"的页表映射方式相比分级页表，有以下的好处和优势：

- 好处：当操作系统启动时，`boot_page_table_sv39` 就采用了一个大页的映射方式，这一方式的好处是能够简便地将操作系统从物理地址的模式切换到虚拟地址模式而不用进行多级映射关系的处理。同时也减少了页表项的数量，节省了内存空间。由于页表项数量减少，"一个大页"的页表映射方式可以提高访问效率，在分级页表中，访问一个虚拟页需要多次查找页表项，而使用"一个大页"的页表映射方式只需要一次查找即可。还可以减少TLB miss，当一个大页被加载到TLB中，可以覆盖更多的虚拟地址范围，减少了TLB缺失的次数，提高了内存访问的效率。

- 坏处：然而，如果在多个进程的情况下，使用一个大页进行映射意味着在发生缺页异常和需要页面置换时需要把整个大页的内容（在 Sv39 下即为 1GiB）全部交换到硬盘上，在换回时也需要将所有的内容一起写回。在物理内存大小不够、进程数量较多而必须要进行置换时，这会造成程序运行速度的降低。还可能会导致内存碎片的问题，内存利用率较低。最后，安全隐患也值得考虑，一个大页泄露更易导致更多信息泄露。

综上，采用"一个大页"的页表映射方式可以减少页表项数量、提高访问效率和减少TLB缺失。然而，它也面临内存碎片、大页分配开销和内存利用效率等问题。

扩展练习Challenge：实现不考虑实现开销和效率的LRU页替换算法

主要修改_lru_tick_event函数，实现LRU算法。这段代码是一个LRU（最近最少使用）算法中的"时钟"算法实现，用于**不考虑实现开销和效率**地更新页面访问情况并调整页面在链表中的位置：

```
// 时钟触发
static int
_lru_tick_event(struct mm_struct *mm)
{
    list_entry_t *head = (list_entry_t *)mm->sm_priv;
    assert(head != NULL);
    list_entry_t *entry = list_next(head);
    while (entry != head)
    {
        struct Page *page = le2page(entry, pra_page_link);
        pte_t *ptep = get_pte(mm->pgdir, page->pra_vaddr, 0);
        // 页表项的A位，即 Accessed，如果 A=1 表示自从上次 A 被清零后，有虚拟地址通过这个页表项进行读、或者写、或者取
        if (*ptep & PTE_A)
        {
            list_del(entry);
            list_add(head, entry);
            *ptep &= ~PTE_A;
            tlb_invalidate(mm->pgdir, page->pra_vaddr);
        }
        entry = list_prev(head);
    }
    return 0;
}
```

可以看出，每个时钟触发时，都会遍历当前管理器中的所有页，然后获取其页表项。如果页表项的A位是1，说明自从上次A位被清零后，有虚拟地址通过这个页表项进行读、或者写、或者取指，也就是说被访问，需要**移至链表的首部**。

对于其他函数，`_lru_map_swappable` **插入时依旧插在链表头部**，`_lru_swap_out_victim` **删除时则应当在尾部删除**。

这样，结合时钟中断的特性触发 `_lru_tick_event` 函数即可达到通过时钟实现的LRU页面置换算法。

附录

练习1完整的描述：

1. `do_pgfault`：首先在发生**pagefault**后，我们的异常处理器会根据情况调用 `do_pgfault` 函数，这也是整个缺页处理流程的开始，根据 `get_pte` 得到的页表项的内容确定页面是需要创建还是需要换入。这个函数在vmm.c文件中定义。
2. `find_vma`：接下来我先依次介绍在 `do_pgfault` 函数中用的关键函数，首先是调用了 `find_vma` 函数找到包含传入的错误地址的VMA，这个函数在vmm.c文件中定义。函数首先尝试使用 `mm` 的 `mmap_cache` 来快速定位是否最近访问的VMA包含这个地址。如果缓存的VMA确实包含该地址，函数就直接使用它。否则，函数会遍历 `mm` 中的所有VMA，一旦找到了符合条件的VMA，则中断。如果找到了VMA，函数会更新 `mmap_cache` 。最终，函数返回找到的VMA或者在未找到时返回NULL。

3. `get_pte`：然后是最重要函数之一： `get_pte` 函数，其会根据触发缺页异常的虚拟地址查找其对应的多级页表的页表项。如果其中某一级页表不存在则创建一个新的页（4KiB）用于存储映射关系。该函数首先通过使用 `PDX1(la)` 宏来索引到所给虚拟地址对应的Giga Page的页表项。如果此页目录条目未被标记为有效（即对应的下一级页表不存在），则会分配一个新的页来存储下一级的页表，并进行适当的初始化。接下来，函数进一步深入到下一个级别的页表，使用类似的逻辑来确保对应的页表存在。最后，函数使用 `PTX(la)` 宏来索引并返回所给虚拟地址的具体页表项的地址，并用 `KADDR` 宏来实现物理到虚拟地址的转换。

4. `pgdir_alloc_page` / `alloc_pages` / `alloc_page`： `pgdir_alloc_page` 函数用来分配和初始化页面的函数，它操作在指定的页目录 `pgdir` 和虚拟地址 `la` 上。其先调用 `alloc_page()` 为给定的虚拟地址分配一个新的物理页面。如果页面分配成功，它会使用 `page_insert()` 函数来将该页面插入到页目录中，并为其设置指定的权限 `perm` 。接着，如果系统的交换空间已经初始化（由 `swap_init_ok` 变量指示），函数会使用 `swap_map_swappable()` 来标记新分配的页面为可交换，并将页面的 `pra_vaddr` 字段设置为指定的虚拟地址。 `alloc_page` 为 `alloc_pages` 的一个宏，会分配**一个页面**，如果不够用则会用 `swap_out` 换出所需的页面数量。

5. `page_insert`：将虚拟地址和新分配的页面的物理地址在页表内建立一个映射。该函数先用 `get_pte` 来检索或分配与给定虚拟地址 `la` 关联的页表项。如果因为内存不足而无法分配新的页表项，函数返回一个错误。接着，它增加指向该物理页面的引用计数，因为新的虚拟地址现在指向它。然后，函数检查所得的页表项是否已经有一个有效的映射（即检查 `PTE_V` 标志）。如果已经存在映射，并且映射到的是同一个页面，函数仅减少页面的引用计数。但如果映射到的是另一个不同的物理页面，函数会使用 `page_remove_pte` 来删除该旧映射。随后，函数使用 `pte_create` 来构造一个新的页表项，将给定的虚拟地址映射到指定的物理页面，并设置所需的权限。最后，用 `tlb_invalidate` 来刷新TLB。

6. `swap_map_swappable`：在使用 FIFO 页面替换算法时会直接调用 `_fifo_map_swappable` ，这会将新加入的页面存入 FIFO 算法所需要维护的队列（使用链表实现）的开头从而保证先进先出的实现。这个函数相当于起到标记这个页面将来是可以再换出的作用，具体的实现就是将该页插到了fifo队列的第一个。

7. `swap_in`：将对应的页面根据存储在页表项位置的 `swap_entry_t` 从“硬盘”中读取页面内容，并且重新写入页面的内存区域。该函数首先调用 `alloc_page` 来分配一个新的物理页面以供使用。在成功分配页面后，它接着通过 `get_pte` 获取与指定的虚拟地址 `addr` 相对应的页表项地址。这个页表项包含了之前页面被换出时的磁盘地址信息。然后，函数利用 `swapfs_read` 从磁盘上读取该页面内容到新分配的物理页面中。在完成读取后，函数打印出相应的调试信息并将新分配的物理页面地址保存在 `ptr_result` 中，以供调用者使用。

8. `swap_out`：根据需要换出的页面数量换出页面到硬盘中。函数接收一个内存管理结构 `mm` 、一个整数 `n` 表示要换出的页面数量，以及一个标记 `in_tick` 。在函数的实现中，它通过一个循环，尝试换出 `n` 个页面。对于每一个要换出的页面，函数首先调用 `sm->swap_out_victim` 来选择一个"受害者"页面，即要被换出的页面。如果成功找到了这样的页面，它会检查页表项确保该页面目前是在物理内存中的。接着，函数尝试通过 `swapfs_write` 将这个受害者页面写到硬盘上。如果写操作成功，函数会更新页表项以反映页面已经被换出，并释放对应的物理页面。此外，为了保证处理器使用的地址转换缓存（TLB）与页表的内容保持一致，函数调用 `tlb_invalidate` 来使得TLB中与该虚拟地址相关的项失效。整体而言， `swap_out` 函数实现了页面置换算法的部分功能，负责将选定的页面从物理内存中换出到磁盘，从而为系统提供更多的可用物理内存空间。

9. `sm->swap_out_victim()`：在 FIFO 中会直接调用 `_fifo_swap_out_victim` 进行页面换出，这会将链表最后（最先进入的页面）指定为需要被换出的页面。这部分实现很简单就是去找到fifo算法维护的那个队列的最后一个，也就是代表最老的页进行换出即可。

10. `free_page` / `free_pages` ：前者是后者的一个宏，会释放一个页面，本质就是调用了 `pmm_manager` 的 `free_pages` 方法将被换出的页面对应的物理页释放从而重新尝试分配。

11. `tlb_invalidate`：在页面换出之后刷新 TLB，防止地址映射和访问发生错误。

12. `swapfs_write`：其会调用 `ide_write_secs` 函数， `ide_write_secs` 的主要目标是模拟向IDE设备（如硬盘）的特定扇区写入数据。它接收四个参数： `ideno` 表示目标IDE设备的编号（这里伪造，实际上就一个）， `secno` 指定要写入的起始扇区号， `src` 是一个指向数据源的指针，而 `nsecs` 表示要写入的扇区数量。函数首先通过 `secno * SECTSIZE` 计算出要写入的数据在IDE设备上的字节偏移量 `iobase` 。接着，它使用 `memcpy` 函数将指定数量的数据从 `src` 复制到模拟IDE设备的存储空间（由数组 `ide` 表示）的相应位置。最后，函数返回0，表示写操作成功完成。简而言之， `ide_write_secs` 函数提供了一个简化的模拟机制，允许用户将数据写入虚拟的IDE设备的特定扇区，而无需实际进行硬盘I/O操作。