Lab3

马浩祎 卢艺晗 陆皓喆

练习1：理解基于FIFO的页面替换算法（思考题）

题目

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

解答

（一）执行流

ucore启动时，会初始化pmm,vmm,swap这些相关的管理员。在程序执行过程中，如果访问到一个不存在的虚拟地址（可能是非法地址，也可能是不在内存中），CPU就会触发缺页异常。操作系统捕获到缺页异常，通过相关csr寄存器获知相关信息（cause指示原因，badvaddr指示出错的虚拟地址），并调用缺页异常处理函数do_pgfault（）。
在do_pgfault()函数中，操作系统首先判断是内存访问越界（无法处理，退出）还是缺页（继续执行）。根据出错的虚拟地址addr找到对应的PTE，将页面内容从磁盘加载到物理内存中，并设置PTE来维护物理地址和虚拟地址的映射。最终，将该page对应的vma加入到mm管理的vma链表中，就是可交换的了，这意味着之后它能够被驱逐出去。
在操作系统将页面内容从磁盘加载到物理内存中之前，首先要pmm为它分配一个物理page。pmm会从free_page链表中寻找空闲页，如果找不到，就只能从当前mm管理着的vma链表中驱逐已被引用的page，释放page资源，这样才能继续加载新的页面内容。

1. 试图访问一个不存在的地址【fifo_check_swap()/swap_fifo.c】
   由于访问的地址无效，CPU触发缺页异常（硬件）并进入内核态
2. 操作系统捕获缺页异常，调用缺页异常处理函数【trap.c】
   1. trap()：派遣异常处理操作者exception_handler(tf)
   2. exception_handler()：根据tf->cause标志位信息，
      在"CAUSE_LOAD_PAGE_FAULT"和"CAUSE_STORE_PAGE_FAULT"两种情况下，会调用缺页异常处理操作员pgfault_handler(tf)
   3. pgfault_handler() : 打印异常信息（print_pgfault()），调用do_pgfault()进行处理
3. 缺页异常处理【do_pgfault()/vmm.c】
   - 寻找虚拟地址addr对应的连续虚拟内存空间【find_vma(mm,addr)/vmm.c】，判断是否是内存访问越界引起的。若是则无法处理，报错退出；否则可以通过取页来处理，继续执行；
   - 寻找虚拟地址addr对应的三级页表项（没有PT则分配,操作失败）【get_pte()/pmmc.】
   - 将所缺页的内容从磁盘加载到物理内存中【swap_in()/swap.c】

- 设置PTE，即维护物理地址和虚拟地址的映射【page_insert()/pmm.c】
- 将该页面设置为可交换的【swap_map_swappable()/swap.c】
- 处理完毕

4. 页面换入与换出关联
    1. 【do_pgfault()/vmm.c】进行缺页异常处理
    2. 【swap_in()/swap.c】将需要的页面从磁盘读入物理内存
    3. 【alloc_page()/pmm.h】加载页面前，先分配一个物理页page
    4. 【swap_out()/swap.c】若分配page失败（可能是没有free_page了），就调用它来换出去一些page（在缺页异常处理中通常n==1）
    5. 【_fifo_swap_out_victim()/swap_fifo.c】sm要换出去一些page，fifo_sm调用响应的驱逐算法，驱逐最早的那个page

（二）函数分析

1. 【find_vma(mm,addr)/vmm.c】
   在指定的交换管理器mm中寻找包含地址addr的一段虚拟内存空间vma；它会优先检查上一次访问的vma（访存局部性原理），若不符合要求，会遍历mm->mmap_list链表来查找；
    - 【list_next(le)/list.h】：链表操作，返回当前链表节点的下一个节点
    - 【le2vma(le,list_link)/vmm.h】：将链表节点转换为对应的vma：调用to_struct()函数，寻找那个list_link成员是当前节点le的vma结构体。
2. 【get_pte()/pmmc.】
   这个函数根据虚拟地址la，结合三级页表寻址的原理，依次提取la的PDX1、PDX0、PDX对应的9位索引，由pgdir指向一级页表的起始虚拟地址，在对应的页表中定位页表项、获取下一级页表地址..... 若该PTE不存在，则为它分配一个物理页，并创建该PTE。最终返回的是三级PTE指针。

```
    /* 取一级PTE指针 */
    pde_t *pdep1 = &pgdir[PDX1(la)];//【PDX1(la)/mmu.h】从虚拟地址la中提取一级页表物理页号（第一个9位）；&pgdir[]根据索引取地址，即取一级PTE的指针

    /* 若该一级页表物理页不存在，为它分配一个物理页*/
    if (!(*pdep1 & PTE_V)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {//【alloc_page()/pmm.h】是宏定义，指向alloc_pages(1)，即为该PT分配一个物理页page
            return NULL;
        }
        set_page_ref(page, 1);//【set_page_ref()/pmm.h】设置页面引用数
        uintptr_t pa = page2pa(page);//【page2pa()/pmm.h】-->【page2ppn()/pmm.h】：物理页page-->物理页帧编号PPN-->物理地址pa
        memset(KADDR(pa), 0, PGSIZE);//将对应的虚拟地址KADDR(pa)处初始化一个页面大小的0
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//创建该PTE，设置type和V标志位
    }
    /* 取二级PTE指针 */
    // ...
    //返回三级PTE指针
```

3. 【swap_in()/swap.c】
   该函数将所缺的页从磁盘读取到物理页中

- 【alloc_page()/pmm.h】分配一个物理页框。是宏定义，指向alloc_pages(1)；
- 【get_pte()/pmm.c】根据虚拟地址，获取对应的三级PTE指针；
- 【swapfs_read()/swapfs.c】从磁盘交换区读取该页的内容到分配的物理page中，是fs封装；
  - 【ide_read_secs()/ide.c】根据指定的磁盘编号，将对应位置的数据复制到dst目的地址中
  - SWAP_DEV_NO：宏定义为1，磁盘编号（本实验就1个）
  - 【swap_offset(entry)/swap.h】：从swap_entry_t（这里是三级PTE）中获取offset那24bits。当缺页时，对应PTE中存放的不再是对应物理页的地址，而是该页在磁盘交换区的偏移。
  - PAGE_NSECT :一个page（4096）占用的磁盘扇区(512)数
  - 【page2kva(page)/pmm.h】根据page结构体获取page的起始虚拟地址

4. 【page_insert()/pmm.c】
   该函数负责设置页表项，构建该虚拟地址la与对应物理页的映射关系
   - 【get_pte()/pmm.c】根据虚拟地址，获取对应的三级PTE指针；
   - 【page_ref_inc/dec(page)/pmm.h】设置物理页page引用数增一/减一
   - 【pte2page(*ptep)/pmm.h】获取ptep指向物理地址的page结构体指针
   - 【page_remove_pte()/pmm.c】解除该虚拟地址la对该物理页的引用（若引用数降为0就释放该page），并刷新TLB将该PTE置为无效
   - 【pte_create()/pmm.c】设置该PTE（包括物理页号、PTE_V位、type）
   - 【tlb_invalidate()/pmm.c】刷新TLB，将TLB中对应的指定PTE置为无效
5. 【swap_map_swappable()/swap.c】
   该函数调用具体的SM的map_swappable()函数，将页面设置为可交换的
   - 【_fifo_map_swappable()/swap_fifo.c】将最近到达的page插入mm管理的vma链表尾部
     - 【list_add()/list.h】将节点添加到链表尾部
6. 【_fifo_swap_out_victim()/swap_fifo.c】
   该函数将最早的page驱逐，即mm管理的vma链表的第一个节点对应page。
   - assert(in_tick==0);确保该函数在非时钟中断的情况下执行
   - 【list_prev()/list.h】链表中取该节点的前驱
   - 【list_del()/list.h】链表中删除指定的节点
   - 【le2page()/memlayout.h】获取该vma链表节点对应的page结构体
7. 【swap_out()/swap.c】
   该函数负责驱逐n个page：对每个page逐一进行 驱逐，获取对应虚拟地址和PTE并验证PTE原本有效，将页面内容写入磁盘交换区（失败则重新设为可交换的），将PTE从标记物理页号改为标记磁盘交换区的位置，并将该page置为free，最后刷新TLB
   - sm->swap_out_victim():调用实际sm的页面驱逐算法
   - get_pte()：由被驱逐页对应的vaddr找到对应的三级PTE
   - 【swapfs_write()/swapfs.c】将被驱逐的page页面内容写入磁盘交换区。
   - sm->map_swappable()：调用实际sm对应的算法，将页面设为可交换的
   - 【free_page()/pmm.h】释放物理页框。该宏指向free_pages(page, 1)，pmm将这一个page放回到空闲列表中
   - tlb_invalidate（）：刷新TLB

8. 【alloc_pages()/pmm.c】
   该函数是pmm的实例化pmm的alloc_pages()的封装接口，它在禁用中断的情况下调用实际pmm的alloc_pages()函数，分配连续n个物理页。若分配成功则返回分配的page，否则驱逐n个page来腾出空间。
   - 【swap_out()/swap.c】见7

函数分析部分分析的函数个数已超过10个，与标号无关。

练习2：深入理解不同分页模式的工作原理（思考题）

题目

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。
- 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？

解答

问题1

1. sv32、sv39、sv48机制的虚拟地址表示异同：
   sv32:

```
+----10----+----10----+----12----+
|  VPN[1]  |  VPN[0]  |  PGOFF   |
+----------+----------+----------+
```

sv39:

```
+----9----+----9----+----9----+----12----+
| VPN[2]  | VPN[1]  | VPN[0]  |  PGOFF   |
+---------+---------+---------+----------+
```

sv48:

```
+----9----+----9----+----9----+----9----+----12----+
| VPN[3]  | VPN[2]  | VPN[1]  | VPN[0]  |  PGOFF   |
+---------+---------+---------+---------+----------+
```

sv32、sv39 和 sv48 的分页机制都使用多级页表实现虚拟地址到物理地址的映射。虚拟地址被分解成多个部分，每个部分用作对应级页表的索引，最后一个部分是物理地址所在物理页的页内偏移。页表查找逐级进行的，从最高级目录开始，每级索引用于查找下一级页表的基地址，直到找到页表项（PTE）。这些不同的机制只在表示位数和索引级数上有所差异，但是逻辑相同。

2. get_pte()函数结构相似性原因：
   在本实验代码中get_pte()的函数实现中，查找一级页表和查找二级页表代码结构十分相似，因为它们执行的操作逻辑是相同的，即：

- 获取虚拟地址相应位置的偏移量，从对应级别的页目录表中索引到该页目录表项，并返回指向该PDX的指针；
- 若该级页目录表的物理页不存在，就为它分配一个物理页；设置页面引用数加一；并在物理页中创建该页目录表项，存储下一级页目录表的物理页号。
  即使是不同的分页机制，在get_pte()函数中的执行逻辑也基本相似，只需要微调部分数值。

问题2

1. 目前这种写法有优点也有劣势：

- **优点:**
  - 简洁。只需调用一个get_pte()函数，就能一次性完成多级页表的逐级访问、分配，并返回最终结果PTE
  - 性能优化。合并在一个函数中，减少了函数调用开销，同时也保证在需要分配时可以立即分配，不会因分离逻辑而增加复杂性或延迟。
- **不足:**
  - 错误处理更复杂。若函数分配页表失败返回NULL，调用者无法区分是一级页表还是二级页表出现问题，在处理时更加麻烦。
  - 只读访问效率更低。当页表项确定存在，调用者只希望获取其内容时，调用get_pte()函数增加了判断分支的开销，效率不是最高。

2. 可以把两个功能分开，建议定义两个函数，一个query_pte()单独用于查询，不做页表存在的判断；一个get_pte()进行逐级页表访问和判断，并在需要时分配。这样两种目的下的调用都能相对简洁高效。

练习3：给未被映射的地址映射上物理页（需要编程）

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。

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 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。
- 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？

设计实现过程

首先提供我编写的代码：

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
```

```
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    //If the addr is in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        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_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    /*
    * 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
    *
    */


    ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                         //PT(Page Table) isn't existed, then
                                         //create a PT.
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
```

```
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {
        /*LAB3 EXERCISE 3: 2211044
        * 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.
            // (1) 尝试加载正确的磁盘页面的内容到内存中的页面
            int result = swap_in(mm, addr, &page); // ***在这里进swap_in函数
            if (result != 0)
            {
                cprintf("swap_in failed\n");
                goto failed;
            }

            // (2) 设置物理地址和逻辑地址的映射
            if (page_insert(mm->pgdir, page, addr, perm) != 0)
            {
                cprintf("page_insert failed\n");
                goto failed;
            }

            // (3) 设置页面为可交换的
            if (swap_map_swappable(mm, addr, page, 1) != 0)
            {
                cprintf("swap_map_swappable failed\n");
                goto failed;
            }

            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
    }
```

```
    ret = 0;
failed:
    return ret;
}
```

主要流程：

- 首先，使用swap_in函数，加载到正确的磁盘页面内容到内存中的页面，再对其进行进一步的判断，确保加载的正确性；
- 然后，通过page_insert函数来设置物理地址和逻辑地址的映射，同样进行对应的判断
- 最后，我们需要设置页面为可交换的，使用swap_map_swappable函数，同时进行判断

回答问题

潜在用处

页目录项（Page Directory Entry）和页表项（Page Table Entry）中的合法位可以用来判断该页面是否存在，还有一些其他的权限位比如可读可写，可以在CLOCK算法或LRU算法中进行调用。

表项中 PTE_A 表示内存页是否被访问过，PTE_D 表示内存页是否被修改过，借助着两位标志位可以实现 **Enhanced Clock 算法。**

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。

页访问异常

具体的跳转流程为：

- 当页访问出现异常的时候，首先根据 `stvec` 的地址跳转到中断处理程序，即 `trap.c` 文件中的 `trap` 函数
- 然后进入到 `exception_handler` 中的 `CAUSE_LOAD_ACCESS` 处理缺页异常
    - ```
      case CAUSE_LOAD_ACCESS:
          cprintf("Load access fault\n");
          if ((ret = pgfault_handler(tf)) != 0) {
              print_trapframe(tf);
              panic("handle pgfault failed. %e\n", ret);
          }
          break;
      ```
- 根据上面的代码可以知道，程序下一步跳转到pgfault_handler函数处
      ```

```
static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}
```

    - 然后只要内存块还有剩余的话，就调用do_pgfault函数处理缺页异常
- 如果处理成功的话，就返回到发生异常处继续执行；如果不成功的话，就输出`unhandled page fault`

对应关系

答：页表项和页目录项存储的结构体如下所示：

```
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;
    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)
};
```

其中我们使用了一个`visited`变量，用来记录页面是否被访问。
在`map_swappable`函数，我们把换入的页面加入到FIFO的交换页队列中，此时页面已经被访问，visited置为1

在`clock_swap_out_victim`，我们根据算法筛选出可用来交换的页面。

- 在CLOCK算法我们使用了visited成员：我们从队尾依次遍历到队头，查看visited变量，如果是0，则该页面可以被用来交换，把它从FIFO页面链表中删除。
- 由于PTE_A表示内存页是否被访问过，visited与其对应。

练习4：补充完成Clock页替换算法（需要编程）

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)

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 比较Clock页替换算法和FIFO算法的不同。

设计实现过程

_clock_init_mm

代码如下所示：

```
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;
}
```

代码步骤

- 我们首先使用list_init初始化pra_list_head为空链表
- 然后令curr_ptr指向表头
- 将mm的私有成员指针sm_priv指向pra_list_head即可

_clock_map_swappable

代码如下所示：

```
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: 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_before(&pra_list_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;
}
```

代码步骤

- 我们使用list_add_before函数，即每次均插到链表头(head指向的链表项的下一个)，之后遍历则从链表尾向前遍历即可
- 然后，将page的是否访问成员赋值为1，表示已经对其进行了访问

_clock_swap_out_victim

代码如下所示:

```
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
        // 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
            curr_ptr = list_next(curr_ptr); // 开始遍历，从当前指针的下一个页面开始
        if (curr_ptr == &pra_list_head)
        {
            // 遍历回到开始，则再次遍历
            curr_ptr = list_next(curr_ptr);
        }
        struct Page *page = le2page(curr_ptr, pra_page_link);

        // 如果页面未被访问
        if (page->visited == 0)
        {
            cprintf("curr_ptr %p\n", curr_ptr);
            // 将该页面从页面链表中删除
            list_del(curr_ptr);

            // 将该页面指针赋值给ptr_page作为换出页面
            *ptr_page = page;

            break;
        }
        else
        {
            // 如果页面已被访问，则将visited标志置为0
            page->visited = 0;
        }
    }
    return 0;
}
```

代码步骤

- 遍历链表，如果下一个指针是`&pra_list_head`，则将其指向为下一个指针。
- 构造页面，判断是否最近被使用过，如果页面未被访问过，则将该页面从链表中删除，然后将页面指针赋值给ptr_page作为换出页面；
- 如果已经访问过了，就将visited标志置为0

比较Clock页替换算法和FIFO算法的不同

回答:

- 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。
- 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。

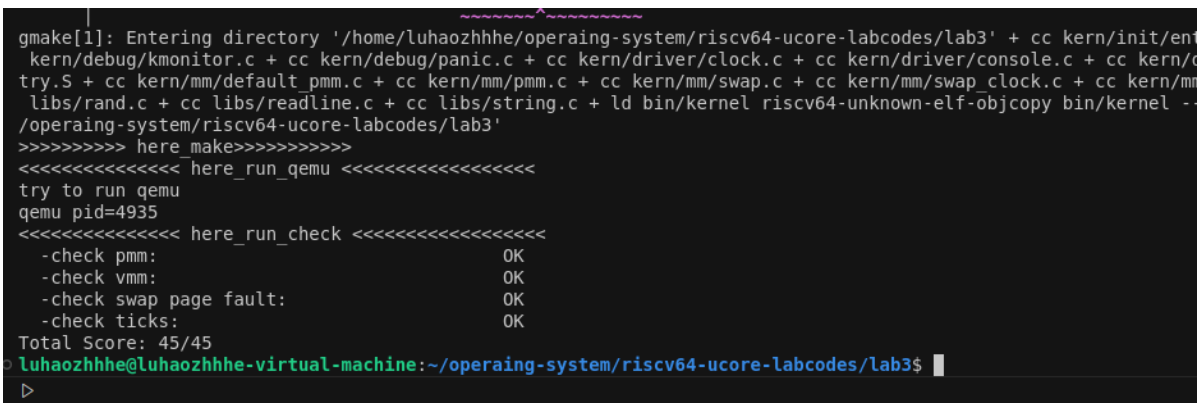
总而言之，CLOCK算法考虑了页表项表示的页是否被访问过，而FIFO不考虑这点。

实验结果

运行

```
make clean
make qemu
make grade
```

得到:



说明实验成功！

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）

问题：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？

回答：

- **优势：**
  - **简单性**：使用一个大页的页表映射方式更为简单和直观。
  - **快速访问**：由于只有一个页表，页表查找速度通常更快，从而可以减少内存访问的延迟。
  - **连续内存分配**：大页可以为需要大量连续内存的应用程序提供更好的性能，因为它们减少了页表条目的数量和TLB缺失的可能性。
  - **减少TLB缺失**：由于大页涵盖的物理内存范围更大，TLB中的一个条目可以映射更大的内存范围，从而可能减少TLB缺失的次数。
- **劣势：**
  - **浪费内存**：如果应用程序只需要小部分的大页，则剩余的部分将被浪费，导致内存碎片。
  - **不灵活**：大页不适合小内存需求的应用程序。
  - **增加内存压力**：由于每个大页都需要大量的连续内存，因此可能会增加内存分配的压力和碎片化。
  - **可能增加页错误**：如果应用程序访问的内存跨越了多个大页，那么可能会导致更多的页错误。
  - **兼容性问题**：不是所有的硬件和操作系统都支持大页。
  - **安全问题**：大页可能会导致更大范围的内存暴露给恶意软件，增加安全风险。

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）

challenge部分不是必做部分，不过在正确最后会酌情加分。需写出有详细的设计、分析和测试的实验报告。完成出色的可获得适当加分。