lab 3

练习0：填写已有实验

本实验依赖实验2。请把你做的实验2的代码填入本实验中代码中有“LAB2”的注释相应部分。（建议手动补充，不要直接使用merge）

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）

流程

check_pgfault() -> 处理缺页异常。 alloc_page() / check_alloc_page() -> 如有需要，分配一个新页面。 fifo_swap_out() -> 根据FIFO策略选择一个页面进行替换。 swap_out() 和 swap_in() -> 将选中的页面换出，并换入所需的页面。 setup_page_table() -> 使用新映射更新页表。

处理

1）缺页异常处理：

当发生缺页异常时（如QEMU日志中所示），处理该异常的代码会检测到需要加载一个新页面。相关函数/宏：check_pgfault()、page_fault_handler()（或者其他用于检测和处理缺页异常的函数）。

2）页面分配：

如果需要加载页面，内存管理器可能会分配一个新页面，或确定可以被换出的页面。 相关函数/宏：check_alloc_page()、alloc_page()，或用于页面分配的宏，如PAGE_ALLOC。

3）FIFO队列更新：

在FIFO策略中，被替换的页面通常是队列最前面的页面。这涉及将最老的页面移出队列，并将新页面加入队列尾部。 相关函数/宏：enqueue_page()、dequeue_page()、fifo_swap_out()（用于维护FIFO队列和选择被替换页面的函数）。

4）换入和换出操作：

当FIFO队列前端的页面被选为替换页面时，系统会将其换出内存以腾出空间。 相关函数/宏：swap_in()、swap_out()、fifo_swap_in()、fifo_swap_out()，或其他与换页管理相关的函数。

5）页表更新：

换出页面后，需要更新页表以反映虚拟地址的新映射关系。 相关函数/宏：setup_page_table()或其他用于更新页表的函数。

6）错误处理与成功检查：

各种函数或宏可能会检查操作是否成功或出现错误，例如check_vmm()、check_swap()用于验证虚拟内存管理和换页操作的成功。

用简单的一两句话描述每个函数在过程中做了什么？(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,完整地体现"从换入到换出"的过程)

从分配开始：

alloc_page() - 分配一个新的物理页面，如果没有可用页面，则可能触发页面置换。

到换出：

fifo_swap_out() - 在FIFO队列中选择最早进入的页面进行替换，实现了页面置换的核心逻辑。

swap_out() - 将选定的旧页面从物理内存中移除，并可能将其数据写入到磁盘上，以便释放空间给新页面。

setup_page_table() - 更新页表，以映射新页面到虚拟地址空间中，同时撤销旧页面的映射。

到换入： swap_in() - 将新页面从磁盘（如果之前被换出）加载到内存中，确保数据可用性。

练习2：深入理解不同分页模式的工作原理（思考题）

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。

get_pte() 负责在页表中查找或创建对应于特定虚拟地址的页表项（PTE）。这些不同的分页模式（sv32, sv39, sv48）指的是页表的层次深度和地址空间大小，其中数字（32, 39, 48）表示虚拟地址的位数。

由于 get_pte() 需要能够适应不同层次的页表结构，因此其中的代码需要能够重复处理不同层次的页表解析。这两段形式类似的代码，可能分别处理不同级别的页表： 访问每级页表时，第一段代码可能负责解析较高级的页表（如对于sv48，可能是第一或第二级），而第二段代码处理更接近物理页的级别（如第三或第四级）。每次代码段的工作都是从当前级别的页表中提取下一级的页表地址，然后访问该地址。并且错误处理和页表项创建：如果在某一级页表中没有找到有效的页表项（即页表项不存在或无效），代码需要能够创建新的页表或页表项。

在不同的分页模式下，get_pte() 需要灵活处理从 2 级到 4 级不等的页表深度。因此，函数内部的代码需要在每个级别重复类似的查找和/或创建页表项的过程，以确保可以针对不同的虚拟地址配置正确的物理地址。

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？

优点

简化代码管理：将查找和分配逻辑集中在一个函数中，可以减少代码重复，简化对页表逻辑的管理。

性能优化：在进行页表项查找时，直接在查找失败时进行分配，可以减少一些重复的查找过程，从而可能提高效率。

错误处理集中：错误处理可以集中在一个地方进行，这样可以确保所有相关的错误都会被适当处理，而不会因为分散在多个函数中而遗漏。

事务性操作：由于修改页表需要原子性操作（特别是在多核环境下），合并函数可以更容易地维护操作的原子性。

缺点

函数复杂度高：合并查找和分配逻辑会使单个函数较为复杂，可能难以理解和维护。

功能耦合：查找与分配功能的耦合可能导致函数的适用性降低，不够灵活。例如，在只需要查询而不需要分配的场景中，依旧要承载分配的逻辑。

测试难度：高度耦合的代码可能会使得单元测试变得更加困难，因为测试用例需要覆盖更多的功能分支和异常情况。

拆开

我认为是否拆分应该根据具体代码情况和性能需求决定。拆分的主要好处是使得函数的功能更加单一，便于理解和测试，同时也提供了更高的适用性和复用性。

如果系统的设计或实现中存在多种不同的查找和分配需求，或者对性能的优化要求较高，拆分可能比较适合。如果代码难以管理，最好还是不拆分。

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。

```
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;
    }

    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;

    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: 2210585  2210587
        * 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;

           // 尝试从交换空间加载页面到物理内存，失败则直接跳转
               if ((ret = swap_in(mm, addr, &page)) == 0) {
           // 成功加载后，建立物理页与虚拟地址映射
```

```
                page_insert(mm->pgdir, page, addr, perm);

            // 设置页面为可交换，并更新页面地址
                swap_map_swappable(mm, addr, page, 1);
                page->pra_vaddr = addr;
              } else {
                    cprintf("swap_in failed for addr %x\n", addr);
                    goto failed;
             }

           } else {
              cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
              goto failed;
            }
       }
   ret = 0;
failed:
    return ret;
}
```

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。

地址映射：PDE和PTE最基本的作用是存储物理地址映射信息，即它们包含指向物理内存地址或下一级页表的指针。

访问权限：PDE和PTE包含权限位（如读、写、执行权限），这些权限对于操作系统保护内存不被非法访问至关重要。

状态位：例如存在位（Present），表明相应的页面是否在物理内存中；修改位（Dirty），表明页面自被加载以来是否被修改过；访问位（Accessed），表明页面自上次清零以来是否被访问过。这些位对于页替换算法非常有用，因为它们帮助操作系统决定哪些页面最适合被替换。

交换位：在支持交换（Swapping）的系统中，PTE可能包含指向磁盘上交换文件或交换分区的指针，这对于虚拟内存管理至关重要。

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?

生成异常：硬件捕捉到无效的内存访问请求并触发缺页异常。

设置异常信息：处理器在触发异常时会设置相关的寄存器，如在RISC-V中将触发异常的虚拟地址存储在stval寄存器中，错误类型存储在scause寄存器。

跳转到异常处理程序：控制权转移到异常处理程序（如ucore中的缺页处理函数do_pgfault），该程序将解析错误信息，并尝试解决缺页问题，例如通过从磁盘加载缺失的页面到物理内存。

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？

对应关系：每个Page项通过PTE中存储的物理地址与虚拟地址关联。PDE可能指向一个页表，而该页表中的每个PTE指向一个Page结构体所管理的物理页。

用途：这种对应关系允许操作系统有效管理物理内存，例如在进行页面置换时，可以快速查找到哪个虚拟地址映射到哪个物理页，以及该物理页的状态（如是否被修改过，是否可以被替换等）。

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)

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: 2210585  2210587*/
     // 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);
     mm->sm_priv = &pra_list_head;
     curr_ptr = &pra_list_head;
     return 0;
}
```

```
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: 2210585  2210587*/
    // 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_before(head,entry);
    page->visited = 1;
    return 0;
}
```

```
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: 2210585  2210587*/
        // 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_next(curr_ptr);
            continue;
        }
        struct Page * page = le2page(curr_ptr,pra_page_link);
        list_entry_t* next = list_next(curr_ptr);
        if(page->visited != 0)
        {
            page->visited = 0;
            curr_ptr = list_next(curr_ptr);
        }
        else
        {
            cprintf("curr_ptr 0xffffffff%x\n",curr_ptr);
            list_del(curr_ptr);
            *ptr_page = page;
            curr_ptr = next;
            break;
        }
    }
    return 0;
}
```

```
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=5945
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check pmm:                                 OK
  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
Total Score: 45/45
hyf@lcsh:~/桌面/OS/lab3$
```

比较Clock页替换算法和FIFO算法的不同。

Clock算法在很多实际系统中相较于FIFO算法有更好的性能，特别是在处理缓存淘汰和内存管理上更加高效和智能。Clock算法通过考虑页面的访问历史来避免频繁使用的页面被过早替换，而FIFO则完全忽视了这一点。因此，Clock算法通常是更优的选择，特别是在需要较高内存效率的环境中。 Clock页替换算法（时钟或循环缓冲算法） 使用一种循环的方式管理"最近最少使用"(LRU) 的近似情况。它通过维护一个循环链表和一个指针（通常被称为"时钟指针"），指向最老的页面。每个页面有一个访问位（通常在PTE中），Clock算法在遍历时检查此位。如果页面最近被访问（访问位=1），则清除访问位并移动指针到下一页；如果访问位已经是0，那么选中这页作为替换目标。

避免了FIFO算法中可能出现的Belady异常（即页面置换导致缺页率上升的问题）。 实现相对简单，执行效率较高，特别是在页面访问位被硬件自动管理的情况下。但是如果所有页面经常被访问，指针可能需要多次循环才能找到替换页面，这在极端情况下会增加延迟。

FIFO算法维护一个队列，新进的页面添加到队列尾部，替换时总是移除队列头部的页面，即最先进入的页面。算法的核心思想是"先进先出"，最早加载到内存的页面将最先被替换。实现极其简单，易于理解和执行。 所需的记录和更新操作最小，开销较低。但是可能遭受Belady异常，特别是在工作集变动较大的情况下。 不考虑页面的使用频率和重要性，可能导致频繁使用的页面被替换，增加了缺页率和性能损耗。

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？

优点

减少页表项数量：使用大页可以减少需要的页表项数量。对于同样大小的内存空间，大页需要的页表项远少于标准页，这可以降低页表本身所占用的内存空间。

提高TLB效率：转换后备缓冲器（Translation Lookaside Buffer, TLB）是一个缓存虚拟地址到物理地址映射的硬件机制。使用大页由于减少了页表项的数量，可以增加TLB的命中率，从而提高系统的地址转换效率和总体性能。

简化内存管理：大页减少了内存管理的复杂性，因为需要处理的页表项更少，页表结构更简单。

降低页表遍历开销：在多级分页系统中，可能需要多次内存访问才能从页表中找到正确的物理地址。使用大页可以减少这种多级遍历，提高地址转换的速度。

缺点

内存利用率下降：大页可能导致内存浪费。如果应用程序的内存需求无法精确地填满一个或多个大页，那么剩余的未使用内存就会浪费，尤其是在内存碎片较多的场景中。

灵活性降低：大页的使用限制了内存分配的灵活性。由于每个页的大小较大，对于小规模的内存请求处理可能不如使用标准页灵活。

页表配置复杂：虽然管理简单，但大页的配置和维护在某些情况下可能比较复杂，特别是需要考虑应用程序的具体内存使用模式。

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）

我们认为核心在于修改系统的内存管理模块，使得它能够跟踪和替换最近最少使用的页。LRU 算法的核心思想是：当需要替换一页时，总是选择最近最少被访问的页。这通常可以通过维护一个双向链表来实现，链表的头部存放最近访问的页，尾部存放最少访问的页。

1.定义并初始化 `lru_swap_manager`

```
struct swap_manager lru_swap_manager = {
    .init = lru_init,
    .map_swappable = lru_map_swappable,
    .set_unswappable = lru_set_unswappable,
    .swap_out = lru_swap_out,
    .tick_event = lru_tick_event,
};
void lru_init(void) {
    list_init(&lru_list); // 初始化 LRU 链表
}
int lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in) {
    list_entry_t *head = &lru_list;
    list_entry_t *entry = &page->page_link;

    if (list_find(head, entry) == NULL) {
        list_add(head, entry); // 将页加入 LRU 链表头部
    }
    return 0;
}
struct Page *lru_swap_out(struct mm_struct *mm, int *swap_in) {
    list_entry_t *tail = list_prev(&lru_list);
    if (tail != &lru_list) {
        struct Page *page = le2page(tail, page_link);
        list_del(tail); // 从 LRU 链表中移除该页
        return page;
    }
    return NULL;
}
```

- `init` 函数初始化 LRU 相关的资源。
- `map_swappable` 函数用于将页标记为可交换，并将其加入 LRU 链表。
- `set_unswappable` 函数将页标记为不可交换。
- `swap_out` 函数执行实际的页替换操作，从 LRU 链表尾部选择最近最少使用的页。
- `tick_event` 用于处理周期性事件，如老化页的计时器更新。
- 在 `lru_init` 函数中，我们初始化链表，用于跟踪 LRU 页。
- `lru_map_swappable` 用于将某个页标记为可交换页，并将其添加到 LRU 链表头部。这样，最近访问的页总是位于链表头部。
- `lru_swap_out` 实现当需要换出一页时，我们从 LRU 链表的尾部找到最近最少使用的页

2.对其他文件进行修改

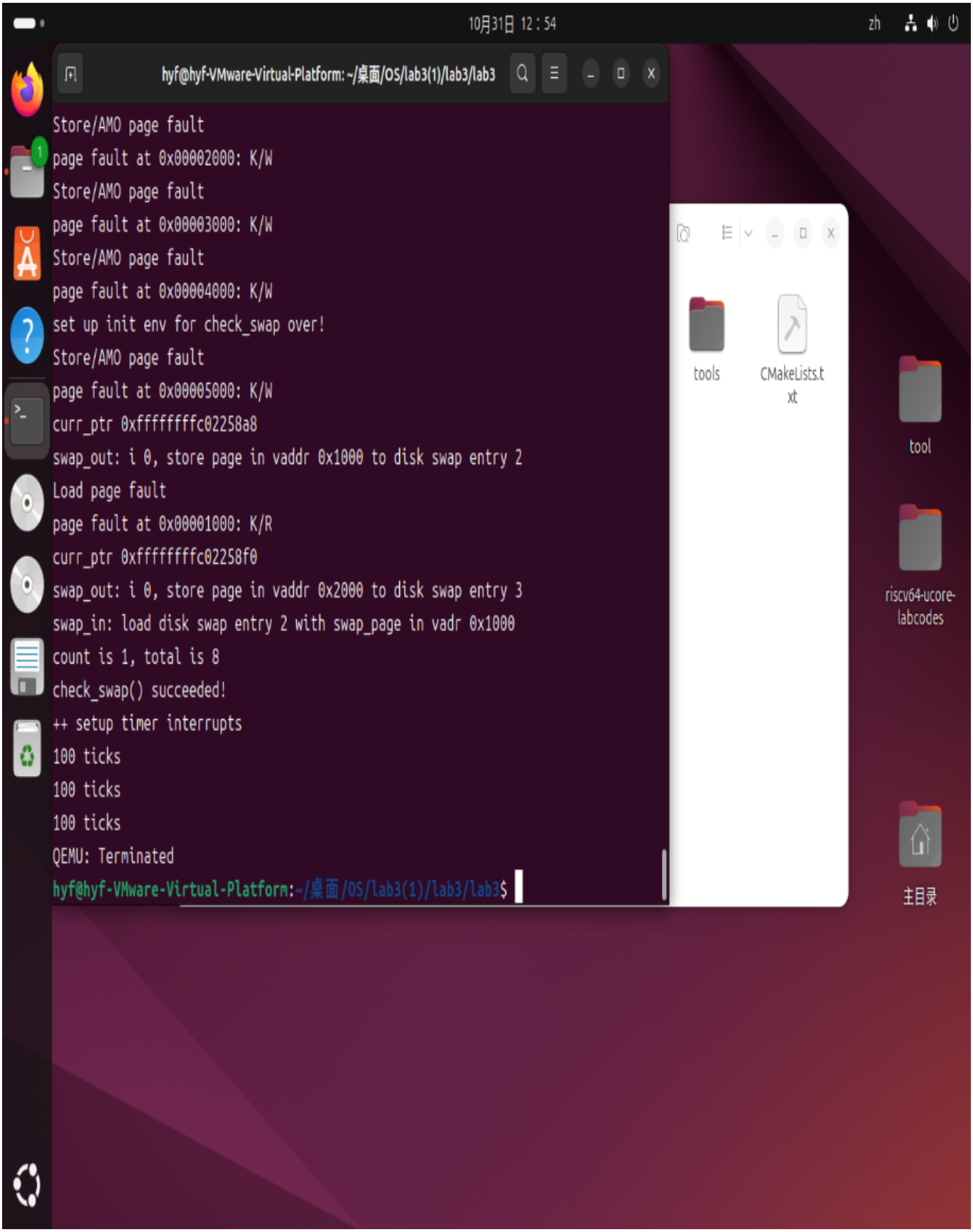
对swap_lru.c以及swap_lru.h进行编写后，主要在进行了以下修改：

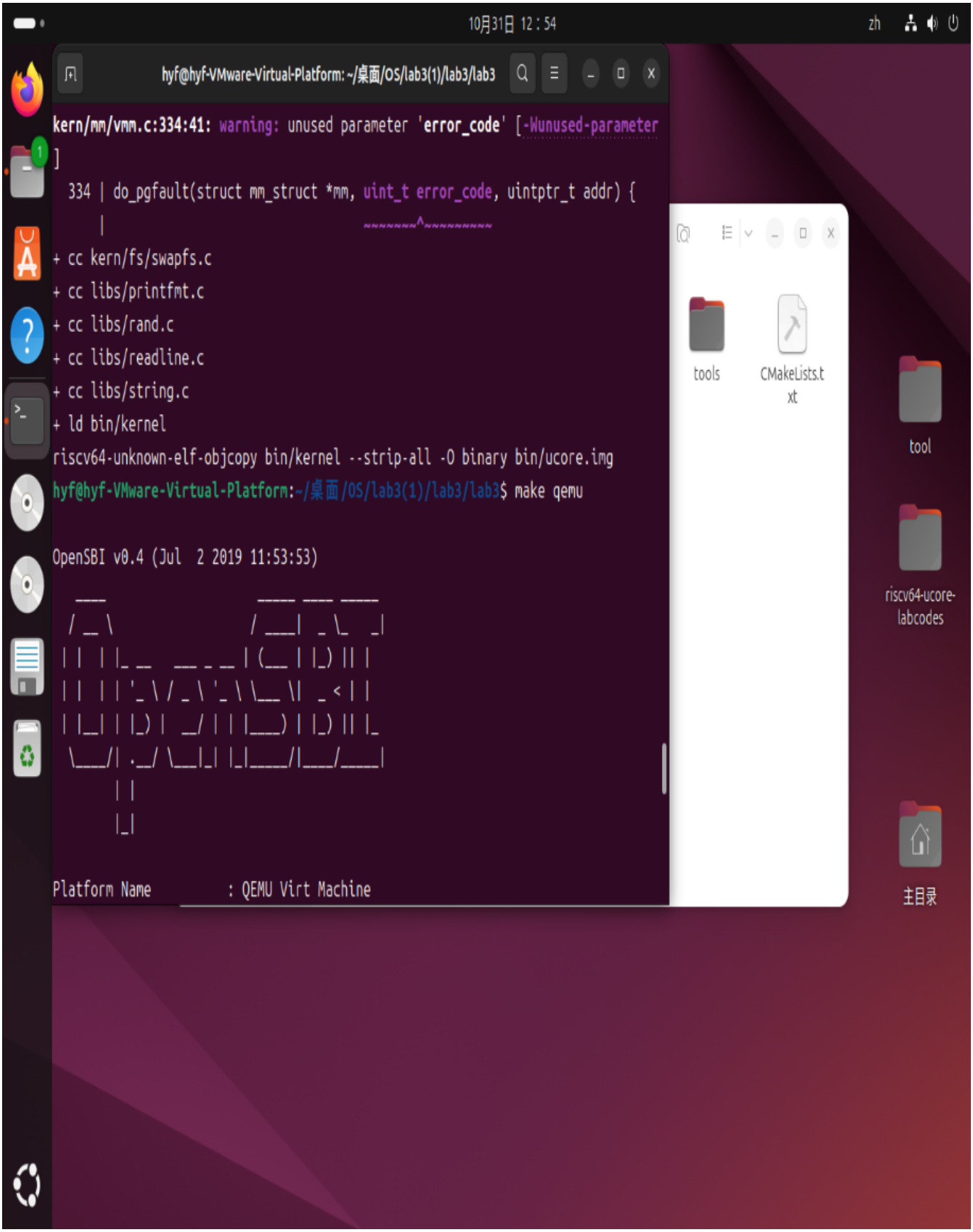
1.在 `kern_init` 函数中调用 `swap_init` 函数来初始化换页系统（`swap_manager`），并确保 `swap_init` 正确地设置为使用 LRU 换页管理器。

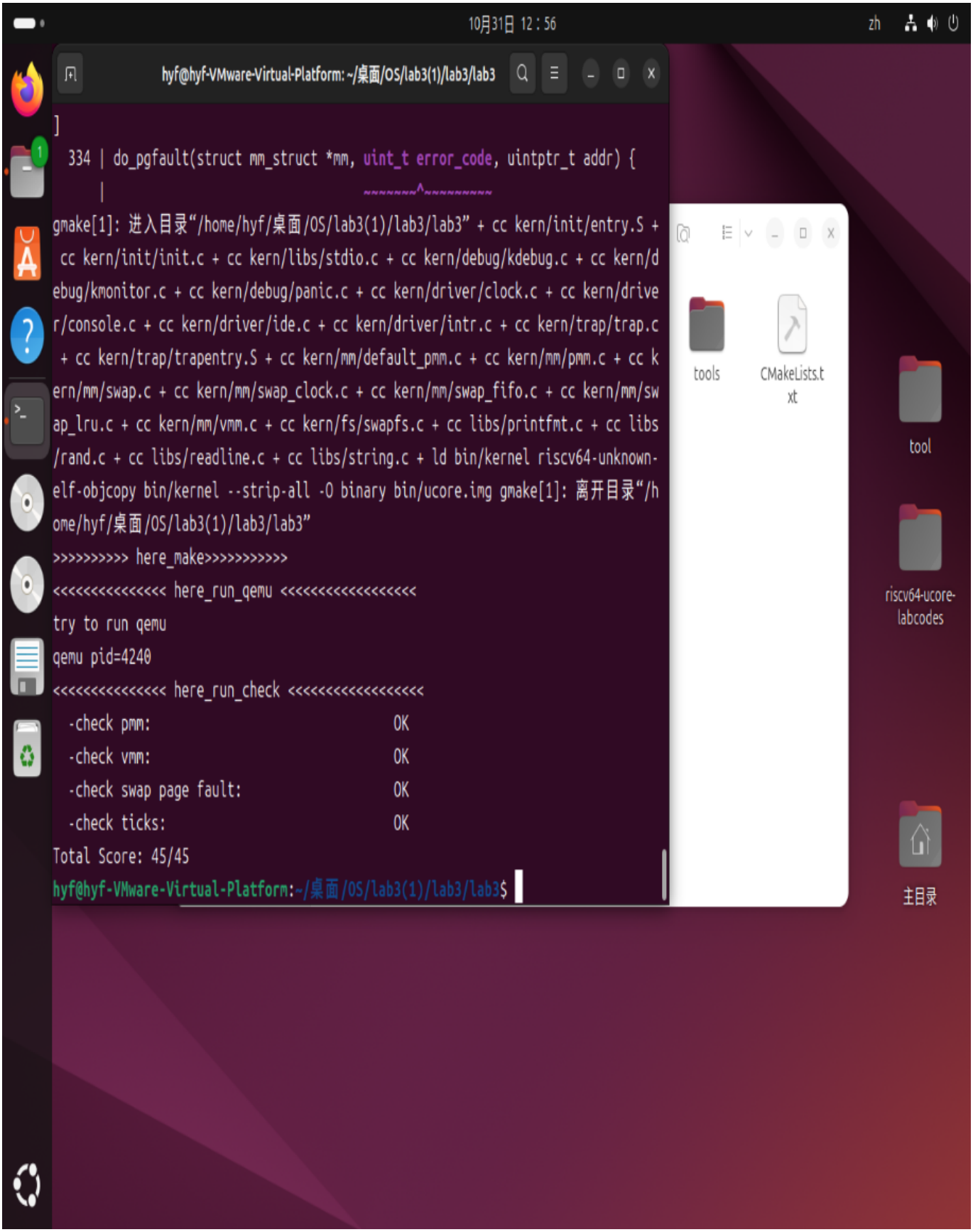
2.在 `swap.h` 中定义 `swap_manager` 结构体指针，并在适当位置引入 `lru_swap_manager` 以支持 LRU。

3.在 `pmm.c` 中增加对 `swap_init_ok` 的引用，以确保在内存不足时启动换页功能。

执行效果


10月31日 12：54
zh
hyf@hyf-VMware-Virtual-Platform: ~/桌面/OS/lab3(1)/lab3/lab3
Store/AMO page fault
page fault at 0x00002000: K/W
Store/AMO page fault
page fault at 0x00003000: K/W
Store/AMO page fault
page fault at 0x00004000: K/W
set up init env for check_swap over!
Store/AMO page fault
page fault at 0x00005000: K/W
curr_ptr 0xffffffffc02258a8
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
Load page fault
page fault at 0x00001000: K/R
curr_ptr 0xffffffffc02258f0
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 1, total is 8
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
QEMU: Terminated
hyf@hyf-VMware-Virtual-Platform:~/桌面/OS/lab3(1)/lab3/lab3$
tools
CMakeLists.txt
tool
riscv64-ucore-labcodes
主目录


10月31日 12：54
zh
hyf@hyf-VMware-Virtual-Platform: ~/桌面/OS/lab3(1)/lab3/lab3
kern/mm/vmm.c:334:41: warning: unused parameter 'error_code' [-Wunused-parameter]
334 | do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
+ cc kern/fs/swapfs.c
+ cc libs/printfmt.c
+ cc libs/rand.c
+ cc libs/readline.c
+ cc libs/string.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
hyf@hyf-VMware-Virtual-Platform:~/桌面/OS/lab3(1)/lab3/lab3$ make qemu
OpenSBI v0.4 (Jul 2 2019 11:53:53)
Platform Name : QEMU Virt Machine
tools
CMakeLists.txt
tool
riscv64-ucore-labcodes
主目录


10月31日 12：56
zh
hyf@hyf-VMware-Virtual-Platform: ~/桌面/OS/lab3(1)/lab3/lab3
]
334 | do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
gmake[1]: 进入目录“/home/hyf/桌面/OS/lab3(1)/lab3/lab3” + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/ide.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/default_pmm.c + cc kern/mm/pmm.c + cc kern/mm/swap.c + cc kern/mm/swap_clock.c + cc kern/mm/swap_fifo.c + cc kern/mm/swap_lru.c + cc kern/mm/vmm.c + cc kern/fs/swapfs.c + cc libs/printfmt.c + cc libs/rand.c + cc libs/readline.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: 离开目录“/home/hyf/桌面/OS/lab3(1)/lab3/lab3”
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=4240
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
-check pmm: OK
-check vmm: OK
-check swap page fault: OK
-check ticks: OK
Total Score: 45/45
hyf@hyf-VMware-Virtual-Platform:~/桌面/OS/lab3(1)/lab3/lab3$
tools
CMakeLists.txt
tool
riscv64-ucore-labcodes
主目录

总结

在实验中，我们主要实现了FIFO、Clock和LRU三种页面替换算法。以下是对各个算法的分析和比较：

1. FIFO算法

FIFO（First In, First Out）算法是一种简单的页面替换算法。该算法通过维护一个队列，将最早进入内存的页面放置在队首，每次发生缺页时将队首页面替换掉。FIFO算法实现步骤如下：

- 新页面进入内存时加入队尾。
- 发生缺页时，将队首页面替换，出列，同时新页面进入队尾。

优缺点：

- **优点**：实现简单，所需额外数据结构少。
- **缺点**：可能导致Belady现象，即增加内存页数反而增加缺页次数，因为最早进入的页面并不一定是最不常使用的页面。

2. Clock算法

Clock算法是一种改进的FIFO算法，引入了访问位（access bit）来决定是否替换页面，从而在一定程度上减少了Belady现象。

- 每个页面都有一个访问位，当页面被访问时，访问位设为1。
- 当发生缺页时，从“时钟”的当前位置开始检查每个页面的访问位。如果访问位为0，则替换该页面，否则将访问位清0并继续检查下一个页面。

优缺点：

- **优点**：相比FIFO更具智能性，能够避免频繁访问的页面被替换。
- **缺点**：实现复杂度稍高，尤其在页面访问频繁时可能需要多次遍历页面。

3. LRU算法

LRU（Least Recently Used）算法通过记录页面最近使用的时间来选择最久未使用的页面进行替换，适用于较少重复访问的页面负载。

- 维护一个页面链表，每次页面被访问时，将其移动到链表尾部。
- 当发生缺页时，替换链表头部页面（即最久未使用的页面）。

优缺点：

- **优点**：理论上最优，因为总是替换掉最近最少使用的页面，适用于大部分工作负载。
- **缺点**：实现复杂度高，需要维护链表或访问时间戳，影响系统性能。