操作系统原理实验

实验四 - 虚拟内存管理

实验准备

主机环境：macOS X 10.14.4 Mojave

虚拟机环境：Linux Ubuntu 14.04 LTS

虚拟机搭载软件：Parallels Desktop.app

命令行终端：Linux 下 Terminal

终端shell：zsh

之前的实验二中，实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好，卡顿严重，分辨率也不高等问题。故从这次实验开始，在Github上将原实验项目clone下来，自己按照指导书第一章讲的环境配置，安装所需支持，配置实用工具进行实验。

为满足实验要求，将命令行用户名字段临时指定为姓名。（需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果，故用我的中大NetID作为署名标记：许滨楠 - xubn。）

实验目的

- 了解虚拟内存的Page Fault异常处理实现；
- 了解页置换算法在操作系统中的实现。

实验内容

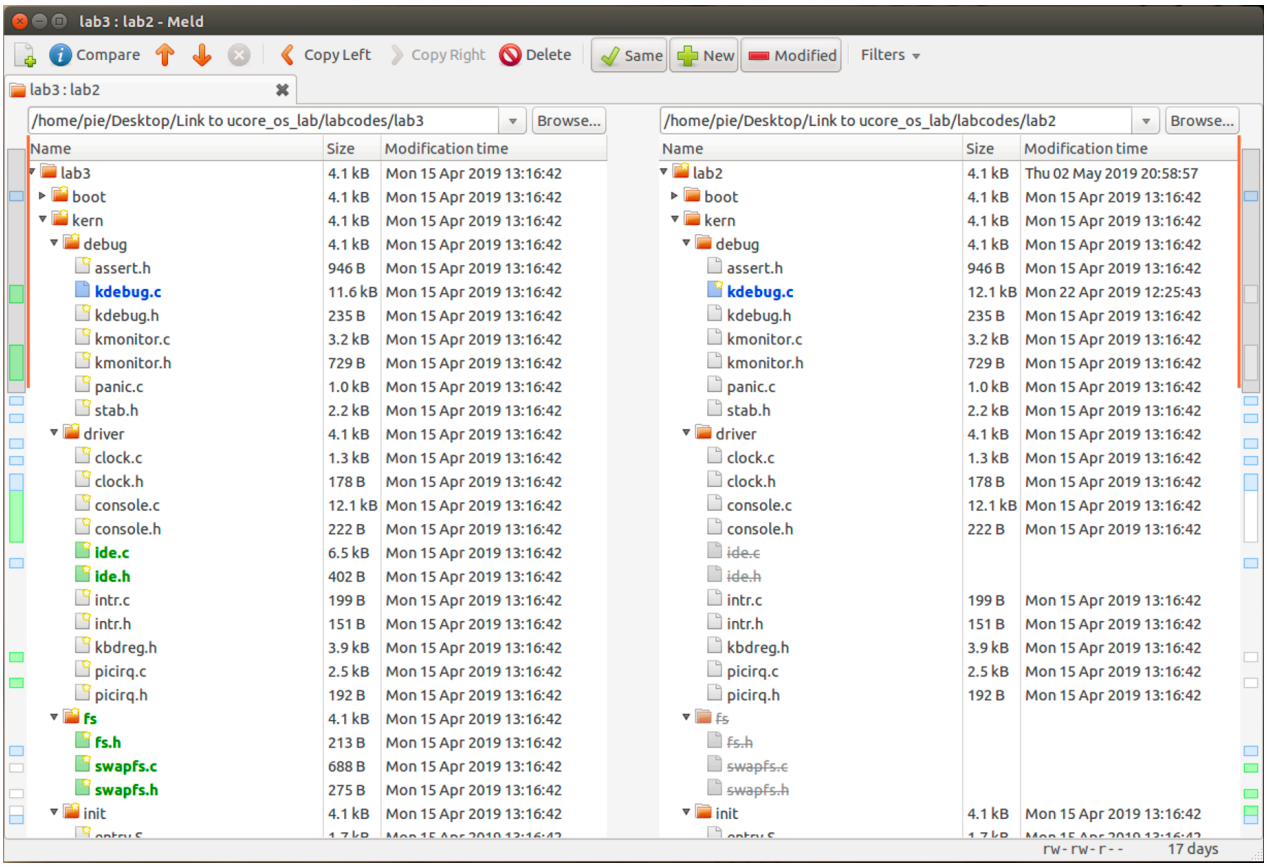
- 在UCORE实验二的基础上，借助页表机制和实验一中涉及的中断异常处理机制，完成Page Fault异常处理和FIFO页替换算法的实现。
- 结合磁盘提供的缓存空间，实现虚存管理，提供一个比实际的物理内存系统更大的虚拟内存系统给系统使用。

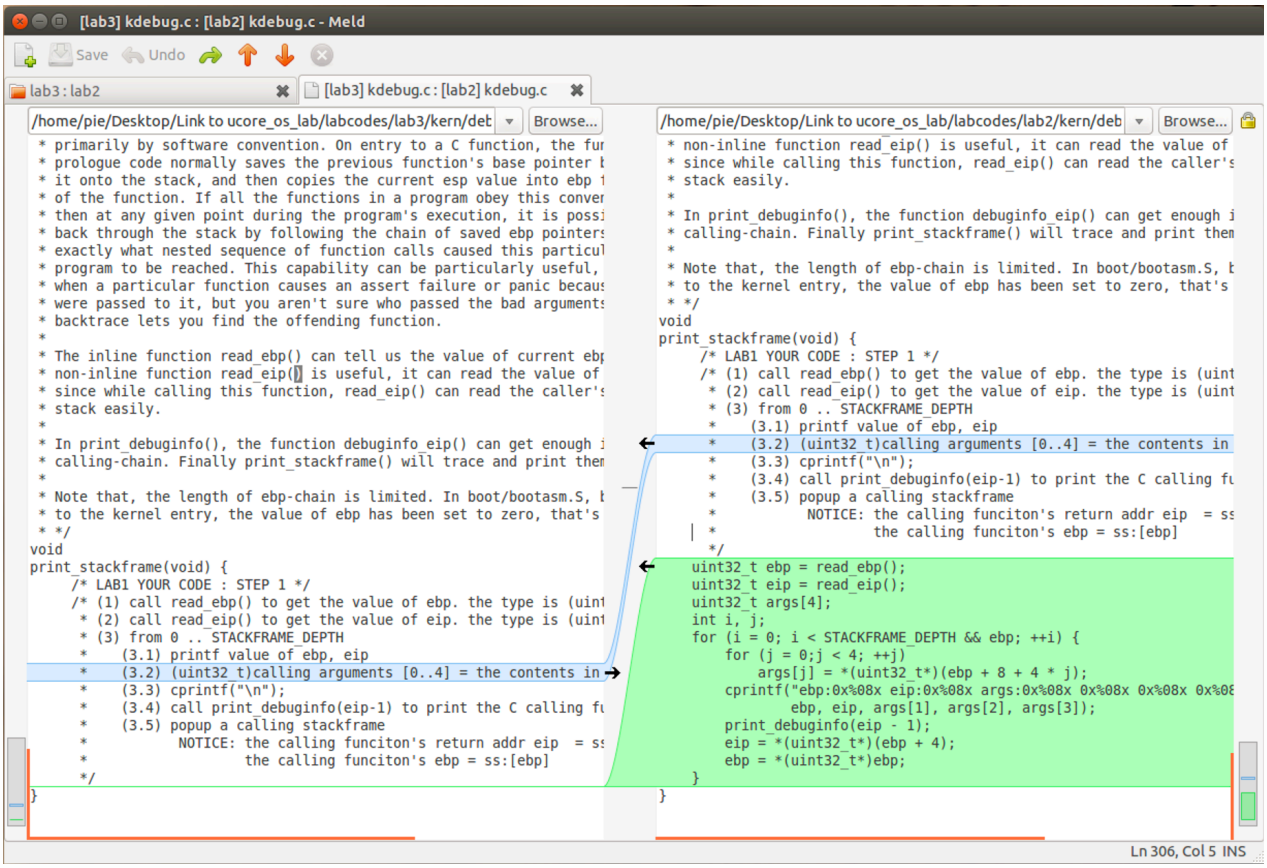
练习

练习0

> 填写已有实验。

利用图形化的merge工具meld，将lab1和lab2中完成的内容填充到lab3中的相应位置。因为之前lab2实验过程中已经将lab1的内容同步更新到lab2中了，所以这次我们只需利用meld的图形化工具，将lab2和lab3整个目录进行对比，定位到相应需要更新的点，将之前完成的代码merge到lab3目录中，方便进行后续实验。





利用merge工具方便地进行代码定位和合并，将lab1和lab2的代码填充完毕后就可以开始正式的实验了。

练习1

> 给未被映射的地址映射上物理页。

基本认识

进入 **/kern/mm/vmm.c** 阅读代码，一些需要了解的关键信息如下。

先是两个最基本的 **manager** 数据结构：

```
/*
  vmm design include two parts: mm_struct (mm) & vma_struct (vma)
  mm is the memory manager for the set of continuous virtual memory
  area which have the same PDT. vma is a continuous virtual memory area.
  There a linear link list for vma & a redblack link list for vma in mm.
*/

// mm_create -  alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));

    if (mm != NULL) {
        list_init(&(mm->mmap_list));
```

```
        mm->mmap_cache = NULL;
        mm->pgdir = NULL;
        mm->map_count = 0;

        if (swap_init_ok) swap_init_mm(mm);
        else mm->sm_priv = NULL;
    }
    return mm;
}

// vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint32_t vm_flags) {
    struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));

    if (vma != NULL) {
        vma->vm_start = vm_start;
        vma->vm_end = vm_end;
        vma->vm_flags = vm_flags;
    }
    return vma;
}
```

根据文件开头的注释说明：对于每个页描述符表 **page descriptor table (PDT)**，有一个 **memory manager (MM)** 负责管理这个PDT的 **virtual memory area (VMA)**，VMA描述一个地址的范围，在PDT中连续地占用地址空间，彼此之间保证不重叠不相交。

对于整个要实现的函数 **do_pgfault** 源码文件中的注释如下：

```
/* do_pgfault - interrupt handler to process the page fault execption
 * @mm         : the control struct for a set of vma using the same PDT
 * @error_code : the error code recorded in trapframe->tf_err which is setted by x86 hardware
 * @addr       : the addr which causes a memory access exception, (the contents of the CR2 register)
 *
 * CALL GRAPH: trap--> trap_dispatch-->pgfault_handler-->do_pgfault
 * The processor provides ucore's do_pgfault function with two items of information to aid in diagnosing
 * the exception and recovering from it.
 *   (1) The contents of the CR2 register. The processor loads the CR2 register with the
 *       32-bit linear address that generated the exception. The do_pgfault fun can
 *       use this address to locate the corresponding page directory and page-table
 *       entries.
 *   (2) An error code on the kernel stack. The error code for a page fault has a format different from
 *       that for other exceptions. The error code tells the exception handler three things:
 *         -- The P flag   (bit 0) indicates whether the exception was due to a not-present page (0)
 *            or to either an access rights violation or the use of a reserved bit (1).
 *         -- The W/R flag (bit 1) indicates whether the memory access that caused the exception
 *            was a read (0) or write (1).
 *         -- The U/S flag (bit 2) indicates whether the processor was executing at user mode (1)
 *            or supervisor mode (0) at the time of the exception.
 */
```

根据注释，**do_pgfault** 函数需要完成的主要有两大块内容：

1. 通过给出的 addr 访问 CR2 寄存器获取其内容定位页目录项和页表项；
2. 根据给出的 error_code 进行错误检查，判断错误类型。

设计实现

函数中已经完成了前面的第二个基本部分，先是通过将页面异常产生的错误码和 3 $(11)_2$ 按位取与，根据低两位，即 P flag 和 W/R flag 判断页面是否存在与内存中，以及对内存区域的读写权限。接下来根据 addr 也就是 CR2 寄存器中存储的内容，需要定位页目录项 page directory entry 和页表 page table，得到对应页表项，然后根据页表项来进行分类讨论：

- 若页表项为0，则系统还未给 addr 分配物理页，这时需要申请分配物理页、设置页目录项和页表，从而建立虚拟地址 addr 到物理页的映射，这也是练习 1 要完成的代码内容；
- 若页表项非0，则系统已经给 addr 分配物理页并建立映射，但是对应的物理页已经被换出到磁盘中，所以这时候也需要分配物理页，把换出的页面内容写到新分配的物理页；
- 最后都需要确保 addr 建立了到物理页的映射，把物理页设置为 swappable，并将其插入可置换物理页链表的尾部，这就是练习2要完成的内容了。

再看练习 1 处的提示注释：

```
    /*LAB3 EXERCISE 1: YOUR CODE
    * 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
    *
    */
#if 0
    /*LAB3 EXERCISE 1: YOUR CODE*/
    ptep = ???              //(1) try to find a pte, if pte's PT(Page Table) isn't existed, then create a PT.
    if (*ptep == 0) {
                            //(2) if the phy addr isn't exist, then alloc a page & map the phy addr with logical addr

    }
```

具体的操作步骤为：

1. 参考 VMA 的权限，根据其来设置物理页的访问权限（源代码中已实现）；
2. 通过虚拟地址 addr 和 PDT 的基址 mm->pgdir，用 get_pte 来得到页表项 PTE 的虚拟地址，如果包含这个页表项的页表不存在，则需要分配物理页、设置权限、建立映射。

具体代码

根据前述的基本认识和设计思路，实现代码如下：

```
// get pte, if it failed stop and inform
if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
    cprintf("do_pgfault failed: get_pte unsuccessfully\n");
    goto failed;
}
// if the physical address does not exist
// alloc a page and map it to the addr
if (*ptep == 0) {
    if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
        cprintf("do_pgfault failed: pgdir_alloc_page unsuccessfully\n");
        goto failed;
    }
}
```

使用 **make qemu** 命令对现在完成的代码进行测试，结果如下：

```
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31994
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
```

输出中有 **check_pgfault() succeeded!** 的输出字样，通过 check_pgfault 函数的测试，代码基本正确。

问题回答

> 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。

页替换算法的实现，要求系统能够区分页的属性：在内存中 / 在外存中 / 不存在，并且在外存中的页还需要进行所在空间的定位。这一点的实现，基于页表项 PTE 。PTE 可以实现页属性的标记并保存页在存储器上的位置。而页目录项 PDE，则是用来索引页表的，所以这两者对实现页置换算法具有重要作用。

> 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?

- 保存页访问异常中断出现之前的进程的CPU现场包括资源管理状态等，将关键信息压栈；
- 根据页访问异常的中断号，跳转到对应的中断服务例程，进入软件层次的例程。

练习2

> 补充完成基于FIFO的页面置换算法。

基本认识

在练习 1 的设计实现中已经讲到，在 **do_pgfault** 函数中，如果物理页不存在且页表项非空，需要用页置换算法将物理页换到内存中，最终完整化 addr 对物理内存页的映射并将该页设置为swappable，这是页面的换入过程，主要在 **do_pgfault** 函数中实现；

如果在将页面调入内存之前，内存已经没有空闲空间了，那么系统需要按照一定的规则将内存中已经有的占用空间的页面换出到外存上，腾出内存空间给新的页面，这就是页面的换出过程，主要在 **swap_fifo.c** 中的 **swap_out_victim** 函数中实现，为了实现这个函数，还要实现其所依赖的另一个函数 **map_swappable** ，这个函数用于记录页面访问情况的相关属性。

设计实现

相应源文件中的代码提示和实现思路如下：

```
/*LAB3 EXERCISE 1: YOUR CODE*/
ptep = ???              //(1) try to find a pte, if pte's PT(Page Table) isn't existed, then create a PT.
if (*ptep == 0) {
                        //(2) if the phy addr isn't exist, then alloc a page & map the phy addr with logical addr

}
else {
/*LAB3 EXERCISE 2: YOUR CODE
* Now we think this pte is a  swap entry, we should load data from disk to a page with phy addr,
* and map the phy addr with logical addr, trigger swap manager to record the access situation of this page.
*
*  Some Useful MACROs and DEFINEs, you can use them in below implementation.
*  MACROs or Functions:
*    swap_in(mm, addr, &page) : alloc a memory page, then according to the swap entry in PTE for addr,
*                               find the addr of disk page, read the content of disk page into this memroy page
*    page_insert ： build the map of phy addr of an Page with the linear addr la
*    swap_map_swappable ： set the page swappable
*/
    if(swap_init_ok) {
        struct Page *page=NULL;
                                //(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.
    }
    else {
        cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
        goto failed;
    }
}
```

对于 **do_pgfault** 函数，当物理页不在内存中、页表项非空且 P flag = 0 时，换入物理面，建立从虚拟地址 addr 到物理页面的映射关系，然后在 map_swappable 中记录下来，维护 page->pra_vaddr ；

```
/*
 * (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
_fifo_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 && head != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 2: YOUR CODE*/
    //(1)link the most recent arrival page at the back of the pra_list_head qeueue.
    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 assign the value of *ptr_page to the addr of this page.
 */
static int
_fifo_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 */
     /*LAB3 EXERCISE 2: YOUR CODE*/
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  assign the value of *ptr_page to the addr of this page
     return 0;
}
```

对于 **swap_fifo.c** 中 **map_swappable** 函数的实现，用链表来实现对 Page 结构的操作，mm 中存放着链表的头节点。通过 Page 中的成员变量 pra_page_link ，按照访问顺序依次插入队首；

对于 **swap_out_victim** 函数的实现，取队列尾部的页面置换，从链表中删除置换的页面。

具体代码

根据注释指引和设计实现中的思路，完成代码如下：

do_pgfault 函数在练习1下方继续补充：

```
else {
    if (swap_init_ok) {
        struct Page *page=NULL;
        // load the content of the corresponding disk page
        // into the memory which page managed
        if ((ret = swap_in(mm, addr, &page)) != 0) {
            cprintf("do_pgfault failed: swap_in unsuccessfully\n");
            goto failed;
        }
        // map the logical addr to the physical address
        page_insert(mm->pgdir, page, addr, perm);
        page->pra_vaddr = addr;
        // make the page swappable.
        swap_map_swappable(mm, addr, page, 1);
    }
    else {
        cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
        goto failed;
    }
}
```

map_swappable 函数完成：

```
static int
_fifo_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 && head != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 2: YOUR CODE*/
    //(1)link the most recent arrival page at the back of the
pra_list_head qeueue.
    list_add_after(head, entry);
    return 0;
}
```

swap_out_victim 函数完成：

```
static int
_fifo_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 */
     /*LAB3 EXERCISE 2: YOUR CODE*/
     //(1)  unlink the  earliest arrival page in front of pra_list_head
qeueue
     //(2)  assign the value of *ptr_page to the addr of this page
     list_entry_t *lastest = head->prev;
     assert(head != lastest);
     struct Page *p = le2page(lastest, pra_page_link);
     assert(p != NULL);
     *ptr_page = p;
     list_del(lastest);
     return 0;
}
```

使用 **make qemu** 命令对现在完成的代码进行测试，结果如下：

```
page fault at 0x00001000: K/R [no page found].
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 0, total is 7
check_swap() succeeded!
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:20:
    EOT: kernel seems ok.
stack trackback:
ebp:0xc011fee0 eip:0xc01009eb args:0xc011ff24 0x00000014 0xc011ff5c 0x00000064
    kern/debug/kdebug.c:309: print_stackframe+21
ebp:0xc011ff10 eip:0xc0100d43 args:0x00000014 0xc01091d8 0x00000000 0x00000064
    kern/debug/panic.c:27: __panic+105
ebp:0xc011ff30 eip:0xc010214b args:0xc0100352 0xc0100304 0xc011ff5c 0x00000064
    kern/trap/trap.c:20: print_ticks+65
ebp:0xc011ff60 eip:0xc0102725 args:0xc0100375 0xc0109084 0xc011ffa4 0x00000064
    kern/trap/trap.c:198: trap_dispatch+204
ebp:0xc011ff80 eip:0xc01027e3 args:0x00000001 0x00000000 0xc011fff8 0x00000064
    kern/trap/trap.c:235: trap+16
ebp:0xc011fff8 eip:0xc01027fb args:0xc0108f28 0xc0100c58 0xc0108f47 0x00000064
    kern/trap/trapentry.S:24: <unknown>+0
Welcome to the kernel debug monitor!!
```

输出中有 **check_swap() succeeded!** 的输出字样，通过 check_pgfault 函数的测试，且100 ticks正常输出显示，代码基本正确。

问题回答

> 如果要在 ucore 上实现 extended clock 页置换算法，请给出你的设计方案，现有的 swap_manager 框架是否足以支持在 ucore 中实现此算法?
>
> > 请给出你的设计方案。（后面将按照在challenge中实现方案，后续分支问题略去）

现有的 swap_manager 框架足以支持在 ucore 中实现 extended clock 页置换算法，具体的实现见扩展练习 challenge 。

扩展练习 Challenge

> 实现识别 dirty bit 的 extended clock 页置换算法。

实验前准备

先利用 git 工具，将当前目录和版本进行备份，推送到远程 github 版本库，同时在本地对 lab3 进行备份，另外建立文件夹进行对 Challenge 的尝试。

```
xubn@ubuntu  labcodes/lab3  master ●  git add .
xubn@ubuntu  labcodes/lab3  master +  git commit -m "2019/5/3 lab3"
[master 42e7b5d] 2019/5/3 lab3
 1 file changed, 4 insertions(+)
xubn@ubuntu  labcodes/lab3  master  git push origin master
Counting objects: 30, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 648 bytes | 0 bytes/s, done.
Total 7 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To git@github.com:PieNam/ucore_os_lab.git
   8299cfb..42e7b5d  master -> master
```

具体思路和实现

"换汤不换药"，利用既成的 FIFO 算法实现的模型和模块功能，对相关源代码进行修改即可开始尝试实现 extended clock 页置换算法。下面利用 **swap_fifo.h** 和 **swap_fifo.c** 的框架，根据需要实现的 extended_clock 进行修改 (ec for extended_clock)，创建 **swap_extended_clock.h** 和 **swap_extended_clock.c** 文件。

对于 **swap_extended_clock.h** 文件，只需要对应修改其全局化的函数名为 ec 版本即可。

对于 **swap_extended_clock.c** 文件，其基本的操作和所要用到的函数都是一致的，主要的几个函数如下：

_ec_init_mm 函数，初始化链表的操作和原来 FIFO 算法是一致的；

_ec_map_swappable 函数，将新的一页插入链表的时候需要进行一些操作上的调整，插入式直接插入到链表尾部，并将其脏位设置为0，采用与 PTE_D 按位取与实现。具体实现函数如下：

```
1  static int
2  _ec_map_swappable(struct mm_struct * mm, uintptr_t addr,
3                    struct Page * page, int swap_in) {
4      list_entry_t *head=(list_entry_t*) mm->sm_priv;
5      list_entry_t *entry=&(page->pra_page_link);
6      assert(entry != NULL && head != NULL);
7      // insert the new page to the back of the list
8      list_add(head->prev, entry);
9      // set its dirty bit to 0
10     struct Page * ptr = le2page(entry, pra_page_link);
11     pte_t * pte = get_pte(mm->pgdir, ptr->pra_vaddr, 0);
12     *pte &= ~PTE_D;
13     return 0;
14 }
```

_ec_swap_out_victim 函数，确定"受害者"，即要被换出的一页，是 extended clock 算法的关键操作，这里采用扫描链表的方式，不断循环检查各页脏位，根据 extended clock 的算法要求：如果 dirty bit 为 1 则设为 0 ，如果 dirty bit 为 0 ，则该页即为将要被换出的一位，除了查找的过程，具体操作仍效仿 FIFO 算法中对应的操作。具体实现函数如下：

```
1  static int
```

```
_ec_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);
    list_entry_t *scanner = head;
    // loop and scan
    while (1) {
        scanner = list_next(scanner);
        if (scanner == head) {
            scanner = list_next(scanner);
        }
        struct Page * ptr = le2page(scanner, pra_page_link);
        pte_t * pte = get_pte(mm->pgdir, ptr->pra_vaddr, 0);
        // if the dirty bit = 1, set it to 0
        if ((*pte & PTE_D) == 1) {
            *pte &= ~PTE_D;
        }
        // else if the dirty bit = 0, set it as the victim
        else {
            assert(ptr != NULL);
            *ptr_page = ptr;
            lisk_del(scanner);
            break;
        }
    }
    return 0;
}
```

测试函数等保留原样，将名称从 FIFO 算法系列改为 extended clock 算法系列(ec)即可；

最后的全局结构体中的字典也需要进行修改，改为 extended clock 算法对应的系列函数，修改后如下：

```
struct swap_manager swap_manager_ec = {
     .name            = "extended_clock swap manager",
     .init            = &_ec_init,
     .init_mm         = &_ec_init_mm,
     .tick_event      = &_ec_tick_event,
     .map_swappable   = &_ec_map_swappable,
     .set_unswappable = &_ec_set_unswappable,
     .swap_out_victim = &_ec_swap_out_victim,
     .check_swap      = &_ec_check_swap,
};
```

最后，为了让 ucore 在运行的时候，使用 extended_clock 算法来替代原来的算法，将调用算法的 **swap.c** 文件中的 **swap_ini** 函数的算法从原先的 FIFO 改为 ec。

利用原来的工具 make qemu 进行测试，发现可以正常工作，运行结果如下：

```
set up init env for check_swap over!
write Virt Page c in ec_check_swap
write Virt Page a in ec_check_swap
write Virt Page d in ec_check_swap
write Virt Page b in ec_check_swap
write Virt Page e in ec_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in ec_check_swap
write Virt Page a in ec_check_swap
page fault at 0x00001000: K/W [no page found].
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
write Virt Page b in ec_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in ec_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in ec_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in ec_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in ec_check_swap
page fault at 0x00001000: K/R [no page found].
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 0, total is 7
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

结果与 FIFO 页置换算法基本一致，考虑到因为样例比较简单， extended clock 算法退化成了 FIFO 算法。扩展练习Challenge完成。

实验结果(练习1, 2, 3)

利用Makefile和Tool中定义的Shell脚本，进行测试：

```
xubn@ubuntu  labcodes/lab3  master  make grade
Check SWAP:                (2.3s)
  -check pmm:                                   OK
  -check page table:                            OK
  -check vmm:                                   OK
  -check swap page fault:                       OK
  -check ticks:                                 OK
Total Score: 45/45
xubn@ubuntu  labcodes/lab3  master  make clean
rm -f -r obj bin
```

练习0、1、2通过基本测试。

扩展练习实验结果在扩展练习板块内已经附上了。

实验总结

分析与区别

- 练习 1 基本与答案一致；
- 练习 2 基本与答案一致，一开始 do_pgfault 函数中自己的实现没有添加对 swap_in 返回值的判定，但后来为了保证程序健壮、考虑到尽可能多的异常因素、防止后续实验中程序更改，还是添加了。错误输出提示也根据答案进行了修改；swap_out_victim 函数中自己的实现原本也没有对 le2page 函数的返回值做安全性的处理，后面页根据答案修改了，并加上了 assert 相关语句；

重要知识点

- 页访问异常的原因、类别、处理方法；
- FIFO实现的页面置换算法；
- 操作系统存储管理。

补充知识点

- 其他许多页面置换算法没有涉及；
- 更多页面标志位的利用。

体会与反思

这一次的实验更加具体化细节化了，主要就是针对虚拟内存的管理这一块内容进行实现，而且最终自己实现/修改的代码也不过四十行。有了前面两次实验的经验，这次的实验做起来更容易明白项目的意思和实验的流程了，许多工具使用起来也更加顺手。整体来讲因为范围的缩小、流程的熟悉、资料的完备，少走了不少弯路。而且也尝试了完成 Challenge ，直接"换汤不换药"地根据原来的 FIFO 算法，对关键的找被置换页的过程进行了修改，如果验证的思路没错的话，这样的操作的确是可行的。所以challenge最终还是大概摸索出来了。

中间还遇到了不小的麻烦，因为Ubuntu系统更新，下载完更新后整个系统的许多包都损坏了，虚拟机显卡驱动也失效，原来的虚拟机不能用了。还好正好是在即将开始尝试 Challenge 的时候，已经先通过 git 版本管理工具将成果推送到远程版本库。花了很多时间重新装了虚拟机配置好环境之后重新完成了实验。（所以有一些截图中可以看出终端主题发生了变化，调不回去了还请见谅！）

以后的实验会更加密集，但应该一样是比较细节化的东西，而不至于像第一次实验那样要对整个过程进行略显盲目的摸索。但也有不少困难的内容，希望能够学好理论知识，在实验中更快理解，也能更进一步熟悉实验的内容和原理。