练习0：填写已有实验

以下操作都是将**内容**复制到 lab3 里，**不要复制整个文件！！**

将 lab1 的 `kern/debug/kdebug.c`、`kern/init/init.c` 以及 `kern/trap/trap.c` 复制到 lab3 里

再将 lab2 的 `kern/mm/pmm.c` 和 `kern/mm/default_pmm.c` 复制到 lab3 里

练习1：给未被映射的地址映射上物理页

do_pgfault 函数源码

写于：**kern/mm/vmm.c**

```
/* 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.
 */
int
do_pgfault(struct mm_struct *mm, uint32_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++;
```

```c
    //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;
    }
    //check the error_code
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        if (!(vma->vm_flags & VM_WRITE)) {
            cprintf("do_pgfault failed: error code flag = write AND not present, but the addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present */
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            cprintf("do_pgfault failed: error code flag = read AND not present, but the addr's vma cannot read or exec\n");
            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_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    /*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

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

根据流程可以知道这个函数是在内核捕获缺页异常之后，通过 IDT 找到的函数，执行该函数来完成缺页异常的处理，先看三个结构体

Page 结构体

写于：**kern/mm/memlayout.h**

```
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as phyical address.
 * */
struct Page {
    int ref;                        // 这个页被页表的引用记数，也就是映射此物理页的虚拟页个数
    uint32_t flags;                 // flags 表示此物理页的状态，1 代表该页是空闲的，0 代表该页已分配
    unsigned int property;          // 记录连续空闲页的数量，只有该页是连续内存块的开始地址时该变量才被使用
    list_entry_t page_link;         // 便于把多个连续内存空闲块链接在一起的双向链表指针(用于物理内存分配算法)
    list_entry_t pra_page_link;     // 便于把多个连续内存空闲块链接在一起的双向链表指针(用于页面置换算法)
    uintptr_t pra_vaddr;            // 这一页的虚拟地址(用于页面置换算法)
};
```

是的又是它，但是它比 Lab2 多了两个变量：`pra_page_link` 和 `pra_vaddr`

mm_struct 结构体

写于：**kern/mm/vmm.c**

```
struct mm_struct {                  // 描述一个进程的虚拟地址空间 每个进程的 pcb 中会有一个指针指向本结构体
    list_entry_t mmap_list;         // 链接同一页目录表的虚拟内存空间中双向链表的头节点
    struct vma_struct *mmap_cache;  // 当前正在使用的虚拟内存空间
    pde_t *pgdir;                   // mm_struct 所维护的页表地址(用来找 PTE)
    int map_count;                  // 虚拟内存块的数目
    void *sm_priv;                  // 记录访问情况链表头地址(用于置换算法)
};
```

vma_struct 结构体

写于：**kern/mm/vmm.c**

```
struct vma_struct {             // 虚拟内存空间
    struct mm_struct *vm_mm;    // 虚拟内存空间属于的进程
    uintptr_t vm_start;         // 连续地址的虚拟内存空间的起始位置和结束位置
    uintptr_t vm_end;
    uint32_t vm_flags;          // 虚拟内存空间的属性 (读/写/执行)
    list_entry_t list_link;     // 双向链表，从小到大将虚拟内存空间链接起来
};
```

find_vma 函数

写于：**kern/mm/vmm.c**

```
// find_vma - find a vma  (vma->vm_start <= addr <= vma_vm_end)
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;  // 初始化 vma 指针变量
    if (mm != NULL) {               // 如果 mm 指针变量不为 0，就进入 if 语句，不然直接返回 NULL
        vma = mm->mmap_cache;       // 将 vma 指针变量的值修改为当前正在使用的虚拟内存空间
        /* 如果当前正在使用的虚拟内存空间不为空，且地址处于正确的 vma 地址内，就不进入 if 语句 */
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
            /* 这下面是处理 vma 异常时的状况的 */
            bool found = 0;                                  // 设立标志位，用于之后确认是否找到符合的 vma
            list_entry_t *list = &(mm->mmap_list), *le = list;  // 这里 *list 和 *le 的值都是 mm->mmap_list
            while ((le = list_next(le)) != list) {           // 没遍历完双向链表就一直遍历
                vma = le2vma(le, list_link);                 // 根据链表找到对应的 vma 结构体基址
                if (vma->vm_start<=addr && addr < vma->vm_end) {// 如果该虚拟地址处在吻合的 vma 地址范围
                    found = 1;                               // 更新标志位为 1，即找到了符合的 vma
                    break;
                }
            }
            if (!found) {                                    // 如果 found 为 0，则没有符合的 vma
                vma = NULL;                                  // 并且更新 vma 为 NULL，之后函数会返回 NULL
            }
        }
        if (vma != NULL) {                                   // vma 不为 NULL 就进入 if 语句
            mm->mmap_cache = vma;                            // 更新当前正在使用的虚拟内存空间为 vma
        }
    }
    return vma;
}
```

le2vma 宏

写于：**kern/mm/vmm.h**

```
#define le2vma(le, member)                  \
    to_struct((le), struct vma_struct, member)
```

和 Lab2 中提到的 le2page 函数基本是一个意思

作用是依靠作为 vma_struct 结构体中 member 成员变量的 le 变量，得到 le 成员变量所对应的 vma_struct 结构体的基地址

ROUNDDOWN 宏

写于：**libs/defs.h**

```
/* *
 * Rounding operations (efficient when n is a power of 2)
 * Round down to the nearest multiple of n
 * */
#define ROUNDDOWN(a, n) ({                                          \
            size_t __a = (size_t)(a);                               \
            (typeof(a))(__a - __a % (n));                           \
        })
```

注释意思：四舍五入操作(当 n 是 2 的幂时有效)，四舍五入到 n 的倍数

其实应该只要 n 不是 0，都可以进行对于 a 的倍数的四舍五入

只是在这个 ucore 的代码里用的都是 2 的倍数（都用的 PGSIZE == 4096）

拿 4 举例的话就是，你能得到：4、8、12、16、20、...

如果 a 是 15 的话，`ROUNDDOWN(15, 4) == 12`

VM_READ 宏

写于：**kern/mm/vmm.h**

```
#define VM_READ                 0x00000001
```

VM_WRITE 宏

写于：**kern/mm/vmm.h**

```
#define VM_WRITE                0x00000002
```

VM_EXEC 宏

写于：**kern/mm/vmm.h**

```
#define VM_EXEC                 0x00000004
```

PGSIZE 宏

写于：**kern/mm/mmu.h**

```
#define PGSIZE          4096                    // bytes mapped by a page
```

E_INVAL 宏

写于：**libs/error.h**

```
#define E_INVAL             3   // Invalid parameter
```

E_NO_MEM 宏

写于：**libs/error.h**

```
#define E_NO_MEM            4   // Request failed due to memory shortage
```

swap_in 函数

写于：**kern/mm/swap.c**

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
    // Page 结构体指针变量 result，result 代表的地址为 alloc_page 申请的页
    struct Page *result = alloc_page();
    assert(result != NULL);                       // 如果 alloc_page 申请页失败了，就中止程序

    pte_t *ptep = get_pte(mm->pgdir, addr, 0);    // 使 ptep 为 PTE 的地址

    int r;
    if ((r = swapfs_read((*ptep), result)) != 0)  // 将硬盘(*ptep)中的内容换入到新的 page(result) 中
    {
        assert(r != 0);                           // swapfs_read 函数的返回值若为 0 则是正常的
    }
    cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n", (*ptep)>>8, addr);
    *ptr_result = result;                         // 更新 *ptr_result 的值为 result
    return 0;
}
```

swapfs_read 函数

写于：**kern/fs/swapfs.c**

```
int
swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT, page2kva(page), PAGE_NSECT);
}
```

SWAP_DEV_NO 宏

写于：**kern/fs/fs.h**

```
#define SWAP_DEV_NO         1
```

swap_offset 宏

写于：**kern/mm/swap.h**

```
/* *
 * swap_offset - takes a swap_entry (saved in pte), and returns
 * the corresponding offset in swap mem_map.
 * */
#define swap_offset(entry) ({                                       \
               size_t __offset = (entry >> 8);                        \
               if (!(__offset > 0 && __offset < max_swap_offset)) {    \
                    panic("invalid swap_entry_t = %08x.\n", entry);    \
               }                                                    \
               __offset;                                            \
          })
```

将传入的地址右移 8 位，再检测其是否满足 swap 的地址范围，满足就返回它

PAGE_NSECT 宏

写于：**kern/fs/fs.h**

```
#define SECTSIZE            512
#define PAGE_NSECT          (PGSIZE / SECTSIZE)
```

已知 PGSIZE 等于 4096，那么 `PAGE_NSECT 就等于 8`

page2kva 函数

写于：**kern/mm/pmm.h**

```
static inline void *
page2kva(struct Page *page) {
    return KADDR(page2pa(page));
}
```

page2pa 的作用是利用 page 这个页的地址找到它所对应的 PPN，也就是物理地址 pa 的前 20 位

KADDR 的作用是通过物理地址找到对应的逻辑(虚拟)地址

`所以 page2kva 函数的作用就是通过物理页获取其内核虚拟地址`

ide_read_secs 函数

写于：**kern/driver/ide.c**

```
int
ide_read_secs(unsigned short ideno, uint32_t secno, void *dst, size_t nsecs) {
    // IDE 硬盘的读函数，参数是 IDE 号，扇区号，缓冲区指针和读扇区个数
    // 一定不能超过最大可读写扇区数，也不能传入无效扇区号
    assert(nsecs <= MAX_NSECS && VALID_IDE(ideno));
    // 传入的扇区号和读取的尾扇区号都不能超出最大扇区数
    assert(secno < MAX_DISK_NSECS && secno + nsecs <= MAX_DISK_NSECS);
    unsigned short iobase = IO_BASE(ideno), ioctrl = IO_CTRL(ideno);
    // 等待磁盘准备好
    ide_wait_ready(iobase, 0);

    // generate interrupt
```

```
    // 向有关寄存器传入 LBA 等参数，准备读
    outb(ioctrl + ISA_CTRL, 0);
    outb(iobase + ISA_SECCNT, nsecs);
    outb(iobase + ISA_SECTOR, secno & 0xFF);
    outb(iobase + ISA_CYL_LO, (secno >> 8) & 0xFF);
    outb(iobase + ISA_CYL_HI, (secno >> 16) & 0xFF);
    outb(iobase + ISA_SDH, 0xE0 | ((ideno & 1) << 4) | ((secno >> 24) & 0xF));
    outb(iobase + ISA_COMMAND, IDE_CMD_READ);

    int ret = 0;
    for (; nsecs > 0; nsecs --, dst += SECTSIZE) {       // 循环读取 nsecs 个扇区
        if ((ret = ide_wait_ready(iobase, 1)) != 0) {    // 出错则 ret 记录错误码，转向 out 返回
            goto out;
        }
        insl(iobase, dst, SECTSIZE / sizeof(uint32_t));  // 向缓冲区读入一个扇区，insl 一次读 32 位
    }
// 如果没有出错，则 ret 保存原值 0，返回
out:
    return ret;
}
```

这具体的以后再看吧

page_insert 函数

写于：**kern/mm/pmm.c**

```
//page_insert - build the map of phy addr of an Page with the linear addr la
// paramemters:
//  pgdir: the kernel virtual base address of PDT
//  page:  the Page which need to map
//  la:    the linear address need to map
//  perm:  the permission of this Page which is setted in related pte
// return value: always 0
//note: PT is changed, so the TLB need to be invalidate
int
page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    pte_t *ptep = get_pte(pgdir, la, 1);        // 获取 pgdir 对应的 ptep
    if (ptep == NULL) {                         // 如果获取 PTE 失败，返回 -4
        return -E_NO_MEM;
    }
    page_ref_inc(page);                         // 将该页的引用计数加 1
    if (*ptep & PTE_P) {                        // 如果 *ptep 有对应的物理地址且存在位为 1
        struct Page *p = pte2page(*ptep);       // 将 p 的值变为 (*ptep) 对应的物理页的地址
        if (p == page) {                        // 如果 p 物理页等于 page 物理页
            page_ref_dec(page);                 // 将该页的引用计数减 1
        }
        else {                                  // 如果 p 物理页不等于 page 物理页
```

```
            page_remove_pte(pgdir, la, ptep);  // 释放 la 虚地址所在的页并取消对应二级页表项的映射
        }
    }
    // 将 page 地址转换为对应的 pa 地址(对应的 PPN 右移 12 位的值)并加上标志位
    *ptep = page2pa(page) | PTE_P | perm;
    tlb_invalidate(pgdir, la);                 // 刷新 TLB
    return 0;
}
```

page_ref_inc 函数

写于：**kern/mm/pmm.h**

```
static inline int
page_ref_inc(struct Page *page) {
    page->ref += 1;
    return page->ref;
}
```

将该页的引用计数加 1

pte2page 函数

写于：**kern/mm/pmm.h**

```
static inline struct Page *
pte2page(pte_t pte) {
    if (!(pte & PTE_P)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));
}
```

先是判断该页的存在位是否为 0，如果为 0，就报错

否则就先利用 PTE_ADDR 将该页的后三位清零，再转化为该物理地址对应的物理页

page_remove_pte 函数

写于：**kern/mm/pmm.c**

```
//page_remove_pte - free an Page sturct which is related linear address la
//                - and clean(invalidate) pte which is related linear address la
//note: PT is changed, so the TLB need to be invalidate
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if ((*ptep & PTE_P)) {
        struct Page *page = pte2page(*ptep);
        if (page_ref_dec(page) == 0) {  // 若引用计数减一后为 0，则释放该物理页
            free_page(page);
        }
        *ptep = 0;                      // 清空 PTE
        tlb_invalidate(pgdir, la);      // 刷新 TLB
    }
```

```
}
```

Lab2 的时候你的练习 3 作业，作用就是**释放某虚地址所在的页并取消对应二级页表项的映射**

swap_map_swappable 函数

写于：**kern/mm/swap.c**

```c
int
swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
{
     return sm->map_swappable(mm, addr, page, swap_in);
}
```

作用就是使这一页可以置换

do_pgfault 函数答案

```c
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    /* *
     * #define E_INVAL             3
     * Invalid parameter
     * */
    int ret = -E_INVAL;
    struct vma_struct *vma = find_vma(mm, addr);  // 试着找到一个包含 addr 的 vma

    pgfault_num++;
    // 如果 addr 不在一个 mm 的 vma 范围内就输出字符串并退出函数，返回值是 3
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    // 检查 error_code
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present 该页不存在 */
        if (!(vma->vm_flags & VM_WRITE)) {  // 验证该页是不是真的可写，不可写就报错
            cprintf("do_pgfault failed: error code flag = write AND not present, but the addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present 该页不可写*/
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present 该页既不可写也不存在*/
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {  // 如果还不能读或者是执行代码，就报错
```

```c
            cprintf("do_pgfault failed: error code flag = read AND not present, but the addr's vma cannot read or exec\n");
            goto failed;
        }
    }

    uint32_t perm = PTE_U;            // perm 代表一个页表的标志位，先使其有用户操作的权限
    if (vma->vm_flags & VM_WRITE) {   // 如果 vma 有可写权限
        perm |= PTE_W;                // 就更新标志位变量，使其也有可写权限
    }
    addr = ROUNDDOWN(addr, PGSIZE);   // 设置 addr 的大小为 4096 的倍数

    ret = -E_NO_MEM;                  // 设置返回值为 -4

    pte_t *ptep = NULL;               // 初始化 PTE 的指针为 NULL

    // try to find a pte, if pte's PT(Page Table) isn't existed, then create a PT.
    // (notice the 3th parameter '1')
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {    // 得到 PTE 的地址，并将其赋给 ptep
        cprintf("get_pte in do_pgfault failed\n");         // 如果没有得到 PTE 的地址，报错
        goto failed;
    }

    // if the phy addr isn't exist, then alloc a page & map the phy addr with logical addr
    if (*ptep == 0) {                                            // 如果 ptep 指针里的物理地址是 0
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {   // 申请一个页并将 ptep 指向新物理地址
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else { // if this pte is a swap entry, then load data from disk to a page with phy addr
           // and call page_insert to map the phy addr with logical addr
        if(swap_init_ok) {                                 // 全局变量，如果 swap 已经完成初始化
            struct Page *page = NULL;                      // 初始化结构体指针变量 page
            // 将硬盘 get_pte(mm->pgdir, addr, 0) 中的内容换入到新的 page 中
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            page_insert(mm->pgdir, page, addr, perm);      // 建立虚拟地址和物理地址之间的对应关系，更新 PTE
            swap_map_swappable(mm, addr, page, 1);         // 使这一页可以置换
            page->pra_vaddr = addr;                        // 设置这一页的虚拟地址，在之后用于页面置换算法
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
```

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

练习2：补充完成基于FIFO的页面替换算法

这里需要更改两个函数 `_fifo_map_swappable` 和 `_fifo_swap_out_victim`，先看两个函数的源码

_fifo_map_swappable 函数源码

写于：**kern/mm/swap_fifo.c**

```
/*
 * (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;
}
```

_fifo_swap_out_victim 函数源码

写于：**kern/mm/swap_fifo.c**

```
/*
 *  (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;
}
```

发现没啥不认识的函数，下面直接放答案

_fifo_map_swappable 函数答案

```
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;  // sm_priv 的作用是记录访问情况链表头地址
    list_entry_t *entry = &(page->pra_page_link);      // 设置 entry 的值为该页的链表地址

    assert(entry != NULL && head != NULL);             // 如果出现地址为 0 的错误就中止程序
    list_add(head, entry);  // 将该页的链表地址(entry)加到链表头节点(mm->sm_priv)的后面
    return 0;
}
```

根据 FIFO 的要求，就是往队首（栈顶）加进去元素，所以这里只需要链表元素加到链表头部后面即可

_fifo_swap_out_victim 函数答案

```
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;  // sm_priv 的作用是记录访问情况链表头地址
    assert(head != NULL && in_tick == 0);         // 如果出现头节点为空或者 in_tick 不为 0 的情况就中止程序
    list_entry_t *le = head->prev;                // 设置 le 为头节点的下一个节点
    assert(head != le);                           // 要是 le 等于 head 就说明双向链表为空，中止程序
    struct Page *p = le2page(le, pra_page_link);  // 通过链表地址找到对应的 Page 结构体
    assert(p != NULL);                            // 没找到 Page 结构体就中止程序
    list_del(le);                                 // 删除双向链表上的 le 节点
    *ptr_page = p;                                // 更新 *ptr_page 为 p
    return 0;
}
```

该函数的作用是选择不需要的页换入硬盘

对于 FIFO 来说，只需要在队尾（栈底）取出一项，即从双向链表中删去链表头部的上一个元素，并将 `*ptr_page` 设置为该页即可

扩展练习 Challenge 1：实现识别dirty bit的 extended clock页替换算法

先咕咕咕，会回来的

扩展练习 Challenge 2：实现不考虑实现开销和效率的LRU页替换算法

同上