

练习0：填写已有实验

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

再将 lab2 的 `kern/mm/pmm.c` 和 `kern/mm/default_pmm.c` 里的内容复制到 lab3 里即可，不要复制整个文件！！

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`

```
1  /* do_pgfault - interrupt handler to process the page fault execption
2   * @mm          : the control struct for a set of vma using the same PDT
3   * @error_code  : the error code recorded in trapframe->tf_err which is
   setted by x86 hardware
4   * @addr       : the addr which causes a memory access exception, (the
   contents of the CR2 register)
5   *
6   * CALL GRAPH: trap--> trap_dispatch-->pgfault_handler-->do_pgfault
7   * The processor provides ucore's do_pgfault function with two items of
   information to aid in diagnosing
8   * the exception and recovering from it.
9   * (1) The contents of the CR2 register. The processor loads the CR2
   register with the
10  *      32-bit linear address that generated the exception. The
   do_pgfault fun can
11  *      use this address to locate the corresponding page directory and
   page-table
12  *      entries.
13  * (2) An error code on the kernel stack. The error code for a page
   fault has a format different from
14  *      that for other exceptions. The error code tells the exception
   handler three things:
15  *      -- The P flag (bit 0) indicates whether the exception was due
   to a not-present page (0)
16  *      or to either an access rights violation or the use of a
   reserved bit (1).
17  *      -- The W/R flag (bit 1) indicates whether the memory access
   that caused the exception
18  *      was a read (0) or write (1).
19  *      -- The U/S flag (bit 2) indicates whether the processor was
   executing at user mode (1)
20  *      or supervisor mode (0) at the time of the exception.
21  */
22  int
23  do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
24      int ret = -E_INVAL;
25      //try to find a vma which include addr
26      struct vma_struct *vma = find_vma(mm, addr);
27  }
```

```

28     pgfault_num++;
29     //If the addr is in the range of a mm's vma?
30     if (vma == NULL || vma->vm_start > addr) {
31         cprintf("not valid addr %x, and can not find it in vma\n", addr);
32         goto failed;
33     }
34     //check the error_code
35     switch (error_code & 3) {
36     default:
37         /* error code flag : default is 3 ( W/R=1, P=1): write,
present */
38         case 2: /* error code flag : (W/R=1, P=0): write, not present */
39             if (!(vma->vm_flags & VM_WRITE)) {
40                 cprintf("do_pgfault failed: error code flag = write AND not
present, but the addr's vma cannot write\n");
41                 goto failed;
42             }
43             break;
44         case 1: /* error code flag : (W/R=0, P=1): read, present */
45             cprintf("do_pgfault failed: error code flag = read AND
present\n");
46             goto failed;
47         case 0: /* error code flag : (W/R=0, P=0): read, not present */
48             if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
49                 cprintf("do_pgfault failed: error code flag = read AND not
present, but the addr's vma cannot read or exec\n");
50                 goto failed;
51             }
52     }
53     /* IF (write an existed addr ) OR
54      *   (write an non_existed addr && addr is writable) OR
55      *   (read an non_existed addr && addr is readable)
56      * THEN
57      *   continue process
58      */
59     uint32_t perm = PTE_U;
60     if (vma->vm_flags & VM_WRITE) {
61         perm |= PTE_W;
62     }
63     addr = ROUNDDOWN(addr, PGSIZE);
64
65     ret = -E_NO_MEM;
66
67     pte_t *ptep=NULL;
68     /*LAB3 EXERCISE 1: YOUR CODE
69      * Maybe you want help comment, BELOW comments can help you finish the
code
70      *
71      * Some Useful MACROS and DEFINES, you can use them in below
implementation.
72      * MACROS or Functions:
73      *   get_pte : get an pte and return the kernel virtual address of this
pte for la
74      *               if the PT contains this pte didn't exist, alloc a page
for PT (notice the 3th parameter '1')
75      *   pgdir_alloc_page : call alloc_page & page_insert functions to
allocate a page size memory & setup

```

```

76      *          an addr map pa<--->la with linear address la and the PDT
pgdir
77      * DEFINES:
78      *   VM_WRITE : If vma->vm_flags & VM_WRITE == 1/0, then the vma is
writable/non writable
79      *   PTE_W      0x002                // page table/directory
entry flags bit : writeable
80      *   PTE_U      0x004                // page table/directory
entry flags bit : User can access
81      * VARIABLES:
82      *   mm->pgdir : the PDT of these vma
83      *
84      */
85  #if 0
86      /*LAB3 EXERCISE 1: YOUR CODE*/
87      ptep = ???                //(1) try to find a pte, if pte's PT(Page
Table) isn't existed, then create a PT.
88      if (*ptep == 0) {
89                              //(2) if the phy addr isn't exist, then alloc
a page & map the phy addr with logical addr
90
91      }
92      else {
93          /*LAB3 EXERCISE 2: YOUR CODE
94          * Now we think this pte is a swap entry, we should load data from
disk to a page with phy addr,
95          * and map the phy addr with logical addr, trigger swap manager to
record the access situation of this page.
96          *
97          * Some Useful MACROS and DEFINES, you can use them in below
implementation.
98          * MACROS or Functions:
99          *   swap_in(mm, addr, &page) : alloc a memory page, then according to
the swap entry in PTE for addr,
100         *                               find the addr of disk page, read the
content of disk page into this memroy page
101         *   page_insert : build the map of phy addr of an Page with the
linear addr la
102         *   swap_map_swappable : set the page swappable
103         */
104         if(swap_init_ok) {
105             struct Page *page=NULL;
106                               //(1) According to the mm AND addr,
try to load the content of right disk page
107                               //      into the memory which page
managed.
108                               //(2) According to the mm, addr AND
page, setup the map of phy addr <---> logical addr
109                               //(3) make the page swappable.
110         }
111         else {
112             cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
113             goto failed;
114         }
115     }
116 #endif
117     ret = 0;
118 failed:

```

```

119     return ret;
120 }

```

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

```

1  /* *
2   * struct Page - Page descriptor structures. Each Page describes one
3   * physical page. In kern/mm/pmm.h, you can find lots of useful functions
4   * that convert Page to other data types, such as physical address.
5   * */
6  struct Page {
7      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
8      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
9      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
10     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)
11     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)
12     uintptr_t pra_vaddr; // 这一页的虚拟地址(用于页面置换算法)
13 };

```

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

```

1  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
2      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
3      struct vma_struct *mmap_cache; // 当前正在使用的虚拟内存空间
4      pde_t *pgdir; // mm_struct 所维护的页表地址(用来找 PTE)
5      int map_count; // 虚拟内存块的数量
6      void *sm_priv; // 记录访问情况链表头地址(用于置换算法)
7  };

```

vma_struct 结构体

写于: kern/mm/vmm.c

```

1 struct vma_struct {           // 虚拟内存空间
2     struct mm_struct *vm_mm; // 虚拟内存空间属于的进程
3     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
4     uintptr_t vm_end;
5     uint32_t vm_flags;       // 虚拟内存空间的属性（读/写/执行）
6     list_entry_t list_link;  // 双向链表，从小到大将虚拟内存空间链接起来
7 };

```

find_vma 函数

写于: kern/mm/vmm.c

```

1 // find_vma - find a vma (vma->vm_start <= addr <= vma->vm_end)
2 struct vma_struct *
3 find_vma(struct mm_struct *mm, uintptr_t addr) {
4     struct vma_struct *vma = NULL; // 初始化 vma 指针变量
5     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
6         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
7         /* 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 */
8         if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr))
9         {
10             /* 这下面是处理 vma 异常时的状况的 */
11             bool found = 0;          // 设立标志
12             位，用于之后确认是否找到符合的 vma
13             list_entry_t *list = &(mm->mmap_list), *le = list; // 这里
14             *list 和 *le 的值都是 mm->mmap_list
15             while ((le = list_next(le)) != list) { // 没遍历完双
16                 向链表就一直遍历
17                 vma = le2vma(le, list_link);      // 根据链表找
18                 到对应的 vma 结构体基址
19                 if (vma->vm_start <= addr && addr < vma->vm_end) { // 如果该虚拟
20                     地址处在吻合的 vma 地址范围
21                     found = 1;          // 更新标志位
22                     为 1，即找到了符合的 vma
23                     break;
24                 }
25             }
26             if (!found) {              // 如果
27                 found 为 0，则没有符合的 vma
28                 vma = NULL;           // 并且更新
29                 vma 为 NULL，之后函数会返回 NULL
30             }
31             if (vma != NULL) {         // vma 不为
32                 NULL 就进入 if 语句
33                 mm->mmap_cache = vma; // 更新当前正
34                 在使用的虚拟内存空间为 vma
35             }
36             return vma;
37 }

```

le2vma 宏

写于: **kern/mm/vmm.h**

```
1 | #define le2vma(le, member) \
2 |     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**

```
1 | /* *
2 |  * Rounding operations (efficient when n is a power of 2)
3 |  * Round down to the nearest multiple of n
4 |  * */
5 | #define ROUNDDOWN(a, n) ({ \
6 |     size_t __a = (size_t)(a); \
7 |     (typeof(a))(__a - __a % (n)); \
8 | })
```

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**

```
1 | #define VM_READ 0x00000001
```

VM_WRITE 宏

写于: **kern/mm/vmm.h**

```
1 | #define VM_WRITE 0x00000002
```

VM_EXEC 宏

写于: **kern/mm/vmm.h**

```
1 | #define VM_EXEC 0x00000004
```

PGSIZE 宏

写于: **kern/mm/mmu.h**

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

E_INVALID 宏

写于: `libs/error.h`

```
1 | #define E_INVALID 3 // Invalid parameter
```

E_NO_MEM 宏

写于: `libs/error.h`

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

swap_in 函数

写于: `kern/mm/swap.c`

```
1 | int
2 | swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
3 | {
4 |     // Page 结构体指针变量 result, result 代表的地址为 alloc_page 申请的页
5 |     struct Page *result = alloc_page();
6 |     assert(result != NULL); // 如果 alloc_page 申请页失
    败了, 就中止程序
7 |
8 |     pte_t *ptep = get_pte(mm->pgdir, addr, 0); // 使 ptep 为 PTE 的地址
9 |
10 |    int r;
11 |    if ((r = swapfs_read(*ptep, result)) != 0) // 将硬盘(*ptep)中的内容换入
    到新的 page(result) 中
12 |    {
13 |        assert(r != 0); // swapfs_read 函数的返回值
    若为 0 则是正常的
14 |    }
15 |    cprintf("swap_in: load disk swap entry %d with swap_page in vadr
    0x%x\n", (*ptep)>>8, addr);
16 |    *ptr_result = result; // 更新 *ptr_result 的值为
    result
17 |    return 0;
18 | }
```

swapfs_read 函数

写于: `kern/fs/swapfs.c`

```
1 | int
2 | swapfs_read(swap_entry_t entry, struct Page *page) {
3 |     return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
    page2kva(page), PAGE_NSECT);
4 | }
```

SWAP_DEV_NO 宏

写于: kern/fs/fs.h

```
1 | #define SWAP_DEV_NO 1
```

swap_offset 宏

写于: kern/mm/swap.h

```
1 | /* *
2 |  * swap_offset - takes a swap_entry (saved in pte), and returns
3 |  * the corresponding offset in swap mem_map.
4 |  * */
5 | #define swap_offset(entry) ({ \
6 |     size_t __offset = (entry >> 8); \
7 |     if (!(__offset > 0 && __offset < max_swap_offset)) { \
8 |         panic("invalid swap_entry_t = %08x.\n", entry); \
9 |     } \
10 |     __offset; \
11 | })
```

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

```
1 | #define SECTSIZE 512
2 | #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

```
1 | static inline void *
2 | page2kva(struct Page *page) {
3 |     return KADDR(page2pa(page));
4 | }
```

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

```
1 | int
2 | ide_read_secs(unsigned short ideno, uint32_t secno, void *dst, size_t
   | nsecs) {
3 |     // 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
4 |     // 一定不能超过最大可读写扇区数，也不能传入无效扇区号
5 |     assert(nsecs <= MAX_NSECS && VALID_IDE(ideno));
6 |     // 传入的扇区号和读取的尾扇区号都不能超出最大扇区数
```



```

7      assert(secno < MAX_DISK_NSECS && secno + nsecs <= MAX_DISK_NSECS);
8      unsigned short iobase = IO_BASE(ideno), ioctrl = IO_CTRL(ideno);
9      // 等待磁盘准备好
10     ide_wait_ready(iobase, 0);
11
12     // generate interrupt
13     // 向有关寄存器传入 LBA 等参数, 准备读
14     outb(ioctrl + ISA_CTRL, 0);
15     outb(iobase + ISA_SECCNT, nsecs);
16     outb(iobase + ISA_SECTOR, secno & 0xFF);
17     outb(iobase + ISA_CYL_LO, (secno >> 8) & 0xFF);
18     outb(iobase + ISA_CYL_HI, (secno >> 16) & 0xFF);
19     outb(iobase + ISA_SDH, 0xE0 | ((ideno & 1) << 4) | ((secno >> 24) &
0xF));
20     outb(iobase + ISA_COMMAND, IDE_CMD_READ);
21
22     int ret = 0;
23     for (; nsecs > 0; nsecs --, dst += SECTSIZE) {           // 循环读取 nsecs 个
扇区
24         if ((ret = ide_wait_ready(iobase, 1)) != 0) {       // 出错则 ret 记录错
误码, 转向 out 返回
25             goto out;
26         }
27         insl(iobase, dst, SECTSIZE / sizeof(uint32_t)); // 向缓冲区读入一个扇
区, insl 一次读 32 位
28     }
29     // 如果没有出错, 则 ret 保存原值 0, 返回
30 out:
31     return ret;
32 }

```

这具体的以后再看吧

page_insert 函数

写于: kern/mm/pmm.c

```

1  //page_insert - build the map of phy addr of an Page with the linear addr
   la
2  // paramenters:
3  // pgdir: the kernel virtual base address of PDT
4  // page: the Page which need to map
5  // la: the linear address need to map
6  // perm: the permission of this Page which is setted in related pte
7  // return value: always 0
8  //note: PT is changed, so the TLB need to be invalidate
9  int
10 page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
11     pte_t *ptep = get_pte(pgdir, la, 1); // 获取 pgdir 对应的 ptep
12     if (ptep == NULL) { // 如果获取 PTE 失败, 返回 -4
13         return -E_NO_MEM;
14     }
15     page_ref_inc(page); // 将该页的引用计数加 1
16     if (*ptep & PTE_P) { // 如果 *ptep 有对应的物理地址且
存在位为 1
17         struct Page *p = pte2page(*ptep); // 将 p 的值变为 (*ptep) 对应的
物理页的地址

```

```

18         if (p == page) {                                // 如果 p 物理页等于 page 物理页
19             page_ref_dec(page);                          // 将该页的引用计数减 1
20         }
21         else {                                            // 如果 p 物理页不等于 page 物理
页
22             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
23         }
24     }
25     // 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
26     *ptep = page2pa(page) | PTE_P | perm;
27     tlb_invalidate(pgdir, la);                          // 刷新 TLB
28     return 0;
29 }

```

page_ref_inc 函数

写于: kern/mm/pmm.h

```

1 static inline int
2 page_ref_inc(struct Page *page) {
3     page->ref += 1;
4     return page->ref;
5 }

```

将该页的引用计数加 1

pte2page 函数

写于: kern/mm/pmm.h

```

1 static inline struct Page *
2 pte2page(pte_t pte) {
3     if (!(pte & PTE_P)) {
4         panic("pte2page called with invalid pte");
5     }
6     return pa2page(PTE_ADDR(pte));
7 }

```

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

```

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

```

```

9         free_page(page);
10    }
11    *ptep = 0;           // 清空 PTE
12    tlb_invalidate(pgdir, 1a); // 刷新 TLB
13 }
14 }

```

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

```

1 int
2 swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
3 int swap_in)
4 {
5     return sm->map_swappable(mm, addr, page, swap_in);
6 }

```

作用就是使这一页可以置换

do_pgfault 函数答案

```

1 int
2 do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
3     /* *
4      * #define E_INVALID          3
5      * Invalid parameter
6      * */
7     int ret = -E_INVALID;
8     struct vma_struct *vma = find_vma(mm, addr); // 试着找到一个包含 addr 的
9     vma
10     pgfault_num++;
11     // 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
12     if (vma == NULL || vma->vm_start > addr) {
13         cprintf("not valid addr %x, and can not find it in vma\n", addr);
14         goto failed;
15     }
16     // 检查 error_code
17     switch (error_code & 3) {
18     default:
19         /* error code flag : default is 3 ( W/R=1, P=1): write, present
20         */
21         case 2: /* error code flag : (W/R=1, P=0): write, not present 该页不存在
22         */
23             if (!(vma->vm_flags & VM_WRITE)) { // 验证该页是不是真的可写，不可写就报错
24                 cprintf("do_pgfault failed: error code flag = write AND not
25                 present, but the addr's vma cannot write\n");
26                 goto failed;
27             }
28             break;
29         case 1: /* error code flag : (W/R=0, P=1): read, present 该页不可写*/
30             cprintf("do_pgfault failed: error code flag = read AND present\n");
31         }
32     }
33 }

```

```

28         goto failed;
29         case 0: /* error code flag : (W/R=0, P=0): read, not present 该页既不可写
也不存在*/
30             if (!(vma->vm_flags & (VM_READ | VM_EXEC))) { // 如果还不能读或者是执
行代码, 就报错
31                 cprintf("do_pgfault failed: error code flag = read AND not
present, but the addr's vma cannot read or exec\n");
32                 goto failed;
33             }
34         }
35
36         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
37         if (vma->vm_flags & VM_WRITE) { // 如果 vma 有可写权限
38             perm |= PTE_W; // 就更新标志位变量, 使其也有可写权限
39         }
40         addr = ROUNDDOWN(addr, PGSIZE); // 设置 addr 的大小为 4096 的倍数
41
42         ret = -E_NO_MEM; // 设置返回值为 -4
43
44         pte_t *ptep = NULL; // 初始化 PTE 的指针为 NULL
45
46         // try to find a pte, if pte's PT(Page Table) isn't existed, then
create a PT.
47         // (notice the 3th parameter '1')
48         if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) { // 得到 PTE 的地
址, 并将其赋给 ptep
49             cprintf("get_pte in do_pgfault failed\n"); // 如果没有得到 PTE
的地址, 报错
50             goto failed;
51         }
52
53         // if the phy addr isn't exist, then alloc a page & map the phy addr
with logical addr
54         if (*ptep == 0) { // 如果 ptep
指针里的物理地址是 0
55             if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) { // 申请一个页
并将 ptep 指向新物理地址
56                 cprintf("pgdir_alloc_page in do_pgfault failed\n");
57                 goto failed;
58             }
59         }
60         else { // if this pte is a swap entry, then load data from disk to a
page with phy addr
61             // and call page_insert to map the phy addr with logical addr
62             if (swap_init_ok) { // 全局变量, 如果
swap 已经完成初始化
63                 struct Page *page = NULL; // 初始化结构体指针
变量 page
64                 // 将硬盘 get_pte(mm->pgdir, addr, 0) 中的内容换入到新的 page 中
65                 if ((ret = swap_in(mm, addr, &page)) != 0) {
66                     cprintf("swap_in in do_pgfault failed\n");
67                     goto failed;
68                 }
69                 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
70                 swap_map_swappable(mm, addr, page, 1); // 使这一页可以置换

```

```

71         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
72     }
73     else {
74         cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
75         goto failed;
76     }
77 }
78 ret = 0;
79 failed:
80     return ret;
81 }

```

练习2：补充完成基于FIFO的页面置换算法

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

```

1  /*
2   * (3)_fifo_map_swappable: According FIFO PRA, we should link the most
recent arrival page at the back of pra_list_head queue
3   */
4  static int
5  _fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
6  {
7      list_entry_t *head=(list_entry_t*) mm->sm_priv;
8      list_entry_t *entry=&(page->pra_page_link);
9
10     assert(entry != NULL && head != NULL);
11     //record the page access situation
12     /*LAB3 EXERCISE 2: YOUR CODE*/
13     //(1)link the most recent arrival page at the back of the pra_list_head
queue.
14     return 0;
15 }

```

`_fifo_swap_out_victim` 函数源码

写于：kern/mm/swap_fifo.c

```

1  /*
2   * (4)_fifo_swap_out_victim: According FIFO PRA, we should unlink the
earliest arrival page in front of pra_list_head queue,
3   *
then assign the value of *ptr_page to the
addr of this page.
4   */
5  static int
6  _fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int
in_tick)
7  {
8      list_entry_t *head=(list_entry_t*) mm->sm_priv;

```

```

9         assert(head != NULL);
10        assert(in_tick==0);
11        /* Select the victim */
12        /*LAB3 EXERCISE 2: YOUR CODE*/
13        //(1) unlink the earliest arrival page in front of pra_list_head
queueue
14        //(2) assign the value of *ptr_page to the addr of this page
15        return 0;
16    }

```

发现没啥不认识的函数，下面直接放答案

fifo_map_swappable 函数答案

```

1  static int
2  _fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
3  {
4      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
5      list_entry_t *entry = &(page->pra_page_link);      // 设置 entry 的值为该
页的链表地址
6
7      assert(entry != NULL && head != NULL);              // 如果出现地址为 0 的
错误就中止程序
8      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
9      return 0;
10 }

```

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 函数源码

```

1  static int
2  _fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int
in_tick)
3  {
4      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
5      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
6      list_entry_t *le = head->prev;                      // 设置 le 为头节点的下一个节
点
7      assert(head != le);                                  // 要是 le 等于 head 就说明
双向链表为空，中止程序
8      struct Page *p = le2page(le, pra_page_link);      // 通过链表地址找到对应的
Page 结构体
9      assert(p != NULL);                                  // 没找到 Page 结构体就中止程
序
10     list_del(le);                                         // 删除双向链表上的 le 节点
11     *ptr_page = p;                                       // 更新 *ptr_page 为 p
12     return 0;
13 }

```

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