操作系统课程设计实验报告

实验名称： 教学操作系统 ucore 实验

一、 实验目的

Ucore 与前几个实验相比，更考察对页表等操作系统概念的理解，通过补全代码并模拟运行系统内核进行实验，从引导开始，系统性地加深对 OS 的理解。

二、 实验环境

操作系统：Linux 4.13.13，Ubuntu 16.04

编译环境：g++ 5.4.0

三、 实验内容

本次实验我完成了 ucore 的 Lab0 – Lab3。

3.0 Lab0 配置环境

Lab 0 主要为从 GitHub clone ucore 的源代码、配置 qemu 和熟悉 makefile 的用法。示例如下：

3.0.1 下载源代码

在 GitHub 仓库中选取 Download，即可以 zip 形式下载全部实验源代码。

ucore_os_lab-master

ucore_os_lab-master .zip

3.0.2 安装 qemu

采用“**sudo apt-get install qemu**”命令安装，安装成功后会有如下提示：

```
qemu is already the newest version (1:2.5+dfsg-5ubuntu10.24).
```

按照各个实验指导即可使用 qemu 运行操作系统。

MakeFile 的用法见下文实验内容。

3.1 Lab1 系统启动

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。

3.1.1 理解通过 make 生成执行文件的过程

通过执行 **make** 命令和分析 MakeFile 文件进行分析。由于 MakeFile 中定义了变量 V，默认不输出编译生成过程，所以在 **make** 命令后加参数“V=”，重定义后可以显示生成过程。

```
V       := @
#need llvm/cang-3.5+
#USELLVM := 1
# try to infer the correct GCCPREFX
```

```
fred@ubuntu:~/Code/ucore_os_lab-master/labcodes/lab1$ make clean
rm -f -r obj bin
fred@ubuntu:~/Code/ucore_os_lab-master/labcodes/lab1$ make qemu V=
+ cc kern/init/init.c
gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-pro
tector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init
/init.c -o obj/kern/init/init.o
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-pro
tector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs
/stdio.c -o obj/kern/libs/stdio.o
+ cc kern/libs/readline.c
gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-pro
tector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs
/readline.c -o obj/kern/libs/readline.o
```

执行 make 命令，生成并载入内核

通过分析 MakeFile 文件，我认为最终内核 ucore.img 的生成依靠一下步骤：

1. **生成 kernel**

该部分为 OS 的内核部分，融合了 ucore 的功能。

```
# -----------------------------------------------------------------
# kernel

KINCLUDE        += kern/debug/ \
                   kern/driver/ \
                   kern/trap/ \
                   kern/mm/

KSRCDIR         += kern/init \
                   kern/libs \
                   kern/debug \
                   kern/driver \
                   kern/trap \
                   kern/mm

KCFLAGS         += $(addprefix -I,$(KINCLUDE))

$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))

KOBJS   = $(call read_packet,kernel libs)

# create kernel target
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld
```

2. **生成 bootblock**

该过程还需要生成 sign 模块

```
# create bootblock
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
        @echo + ld $@
        $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
        @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
        @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
        @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create_target,bootblock)

# -----------------------------------------------------------------
```

3. **将 kernel 和 bootblock 整合为 ucore.img**

```
# create ucore.img
UCOREIMG        := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)
        $(V)dd if=/dev/zero of=$@ count=10000
        $(V)dd if=$(bootblock) of=$@ conv=notrunc
        $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create_target,ucore.img)

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

$(call finish_all)
```

一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

分析 sign.c 文件，可以得到 512 字节引导信息的结构结尾两字节为 0x55 和 0xAA。其中 sign.c 的功能是将指定信息写入目标引导文件（argv[2]）。

```
buf[510] = 0x55;
buf[511] = 0xAA;
FILE *ofp = fopen(argv[2], "wb+");
size = fwrite(buf, 1, 512, ofp);
```

3.1.2 使用 qemu 执行并调试 lab1 中的软件

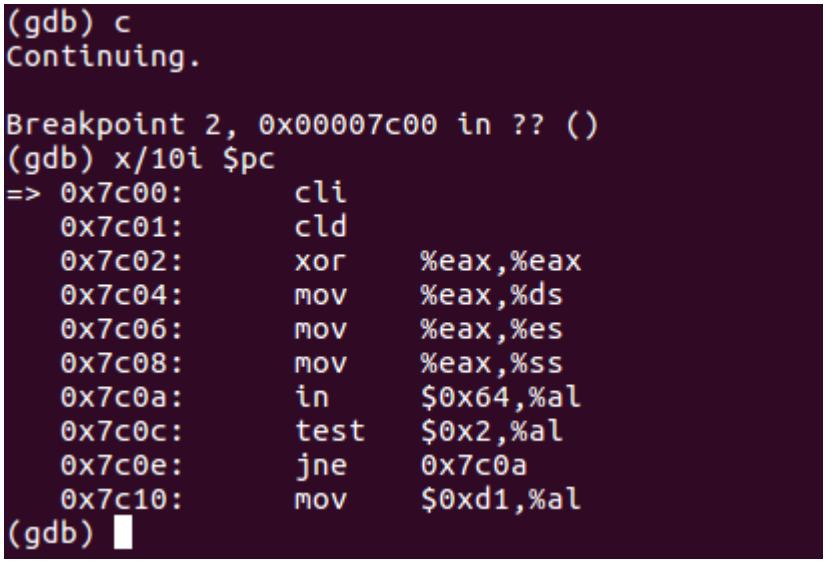
执行 “**make debug-mon**” 命令，进入调试模式，并在 0x7c00 处加入断点。根据 bootasm.S 的描述，0x7c00 为启动的起始地址。

```
(gdb) b*0x7c00
Breakpoint 2 at 0x7c00: file boot/bootasm.S, line 16.
```

命中断点后，检查断点 0x7c00 处的指令和 bootasm.S 的指令，完全一致：

```
# start address should be 0:7c00, in real mode, the beginning address of the
.globl start
start:
.code16                                             # Assemble for 16-bit mo
    cli                                             # Disable interrupts
    cld                                             # String operations incr

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment
```



接下来可以通过 n(next)，c(continue)命令继续调试。

3.1.3 分析 bootloader 进入保护模式的过程

首先使用 cli 命令关闭中断，并将 DS 等寄存器置零。之后开启 A20 全部地址线。在实模式下，20 位地址线寻址空间为 1M，保护模式全部打开后 32 位寻址空间为 4G。

```
seta20.1:
    inb $0x64, %al                              # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                             # 0xd1 -> port 0x64
    outb %al, $0x64                             # 0xd1 means: write data to 8042's P2 port

seta20.2:
    inb $0x64, %al                              # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                             # 0xdf -> port 0x60
    outb %al, $0x60                             # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
```

保护模式的标志为 CR0 寄存器 PE 位为 1，**lgdt** 指令读取 gdt 表。随后跳转至 32 位代码段，调用 bootmain 函数，进入保护模式，切换完毕。

```
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0

    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg

.code32                                             # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax                       # Our data segment selector
    movw %ax, %ds                                   # -> DS: Data Segment
    movw %ax, %es                                   # -> ES: Extra Segment
    movw %ax, %fs                                   # -> FS
    movw %ax, %gs                                   # -> GS
    movw %ax, %ss                                   # -> SS: Stack Segment

    # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain
```

3.1.4 分析 bootloader 加载 ELF 格式的 OS 的过程

bootloader 如何读取硬盘扇区的？

阅读 bootmain()函数，发现函数不断调用 readseg()函数，直至读取至结尾。

```
// load each program segment (ignores ph flags)
ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph ++) {
    readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
}
```

readseg()函数的过程为：调用 waitdisk()函数等待磁盘准备完毕，发起读扇区的命令，再次等待磁盘准备完毕，最后将磁盘数据读到内存。

加载 ELF 的方式为：对于加载可执行文件，遍历 Program Header Table 中的每一项，把每个 Program Header 描述的 Segment 加载到对应的虚拟地址。

3.1.5 实现函数调用堆栈跟踪函数

实现函数如下（kdebug.c）：

```
// get ebp and eip
uint32_t ebp_value, eip_value;
ebp_value = read_ebp();
eip_value = read_eip();
for (int index = 0; index < STACKFRAME_DEPTH; index++) {
        // ebp is the address where saves some arguments.
        uint32_t *addr = (uint32_t *)ebp_value + 2;     // former_ebp(ebp + 0

        cprintf("stackframe info: \n");
        cprintf("ebp: 0x%08x    eip: 0x%08x\n", ebp_value, eip_value);

        for (int i = 0; i < 4; i++) cprintf("arg[%d]: 0x%08x ", i, addr[i]);
        cprintf("\n");

        print_debuginfo(eip_value - 1);
        eip_value = ((uint32_t *)ebp_value)[1];
        ebp_value = ((uint32_t *)ebp_value)[0];
        if (ebp_value == 0) break;
}
```

实现思想为：根据调用子函数，相关参数压栈的顺序，从栈中获取上层函数的 ebp，eip 的值，并获取四个参数。之后跳转至上层函数 ebp 地址，继续迭代输出，完成递归堆栈跟踪。

```
stackframe info:
ebp: 0x00007b38    eip: 0x00100a37
arg[0]: 0x00010094 arg[1]: 0x00010094 arg[2]: 0x00007b68 arg[3]: 0x00100084
    kern/debug/kdebug.c:309: print_stackframe+21
stackframe info:
ebp: 0x00007b48    eip: 0x00100d4d
arg[0]: 0x00000000 arg[1]: 0x00000000 arg[2]: 0x00000000 arg[3]: 0x00007bb8
    kern/debug/kmonitor.c:125: mon_backtrace+10
stackframe info:
ebp: 0x00007b68    eip: 0x00100084
arg[0]: 0x00000000 arg[1]: 0x00007b90 arg[2]: 0xffff0000 arg[3]: 0x00007b94
    kern/init/init.c:48: grade_backtrace2+19
```

3.1.6 完善中断初始化和处理

中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

中断描述符表一个表项占 8 字节。其中 0~15 位和 48~63 位分别为 offset 偏移量的低 16 位和高 16 位。16~31 位为段选择子。通过段选择子获得段基址，加上偏移量即可得到中断处理代码的入口。

完成 idt_init()

```
// define glob variables here:
extern uint32_t __vectors[];
// there are 256 ints.
const size_t int_size = 256;
for (int i = 0; i < int_size; i++) {
    if (i != T_SYSCALL) {
    // init gates, this macro is defined in mmu.h
    SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
}
// to_kernel intr'd dpl should set to DPL_USER
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
lidt(&idt_pd);
cprintf("Ints_init finished.\n");
```

SETGATE 宏的功能为设置 idt 表中的某一个描述符。值得注意的是，T_SWITCH_TOK 中断，即权限提升中断的 DPL 应为用户态，所以要单独设置 idt[T_SWITCH_TOK]。

编程完善 trap.c 中的中断处理函数 trap

修改时钟中断的 case 块即可，内容如下：

```
  case IRQ_OFFSET + IRQ_TIMER:
      /* LAB1 YOUR CODE : STEP 3 */
      /* handle the timer interrupt */
      /* (1) After a timer interrupt, you
ock.c
       * (2) Every TICK_NUM cycle, you can
       * (3) Too Simple? Yes, I think so!
       */
      ticks ++;
      //cprintf(" -- ticks = %d\n", ticks)
      if (ticks % TICK_NUM == 0) {
          print_ticks();
          ticks = 0;
      }
      break;
```

每次产生时钟中断全局变量 ticks 自增，并以 100 为周期输出文字。

```
+++ switch to kernel mode +++
2: @ring 0
2:  cs = 8
2:  ds = 10
2:  es = 10
2:  ss = 10
100 ticks
100 ticks
100 ticks
100 ticks
```

3.1.7 Lab1 总结

本次实验的主要内容是研究内核加载启动。通过实验我了解了编译器是如何将 OS 源代码转

换为真正可以启动的内核的。Lab1 的内容与底层汇编联系较紧密，直观地体现了系统是如何引导的。

3.2 Lab2 物理内存管理

理解基于段页式内存地址的转换机制，理解页表的建立和使用方法，理解物理内存的管理方法。

3.2.1 实现 first-fit 连续物理内存分配算法

1. 首先修改初始化函数 **default_init_memmap()**。

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    //list_add(&free_list, &(base->page_link));

    /* biliteral list, so add_before actually makes it consequent. */
    list_add_before(&free_list, &(base->page_link));
}
```

其功能为为初始化 n 块相连的页，将其加入双向链表。

2. 修改分配页函数 **default_alloc_pages()**。

根据 first-fit 思想，该分配函数从空闲链表中顺序查找，直到寻找到第一个大小满足期待大小的空闲页 page。从 page 中截取 n 块内存页面分配，然后将剩余部分重新加入空闲页面。

```
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;  // alloc n pages in the front, others remain
            p->property = page->property - n;
            SetPageProperty(p);         //
            list_add(&free_list, &(p->page_link));
        }
        //SetPageReserved(page);          // page was used.
        nr_free -= n;
        ClearPageProperty(page);
        list_del(&(page->page_link));   //
    }
    return page;
}
```

3. 修改释放页面函数 default_free_pages()

出于安全性，首先检查 n 块页面的属性是否可以删除（是否为保留）。随后遍历空闲页面链表，寻找合适的位置加入该页面。

```
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    list_add_before(&free_list, &(base->page_link));
```

3.2.2 实现寻找虚拟地址对应的页表项

页面寻址的相关数据结构为：页目录项（一级页表项）pde_t，页表项（二级页表项）pte_t，线性地址 **uintptr_t。**

当已知线性地址 la 时，首先由 pgdir 获取页目录基址。调用 **get_pte()**函数获取页表项。根据线性地址判断在页目录表中是否存在该地址所属的页目录表项，不存在则创建。申请新物理页由 **alloc_page()**函数实现。

根据相关宏定义，将线性地址转换为页表项。

```
pde_t * pdep = &pgdir[PDX(la)];     // get page dir entry (in page dir table) addr.
if ((*pdep & PTE_P) == 0) {           // not PRESENT
    if (!create) return NULL;       // not PRESENT and not create.
    struct Page * page;
    if ((page = alloc_page()) == NULL) return NULL; // failed to create
    set_page_ref(page, 1);
    uintptr_t pa = page2pa(page);
    memset(KADDR(pa), 0, PGSIZE);
    //  now *pdep is invalid
    *pdep = pa | PTE_P | PTE_W | PTE_U;
}
return &((pte_t *)KADDR(PDE_ADDR(* pdep)))[PTX(la)];
```

如果出现访问异常，此函数会返回 NULL，触发缺页中断。

3.2.3 释放某虚地址所在的页并取消对应二级页表项的映射

释放页可以通过 **page_remove()**函数实现，同时根据线性地址 la 取消二级页表项的映射。若该页对应的二级页表项仅有一个（页面的引用次数为 1），则释放页面。

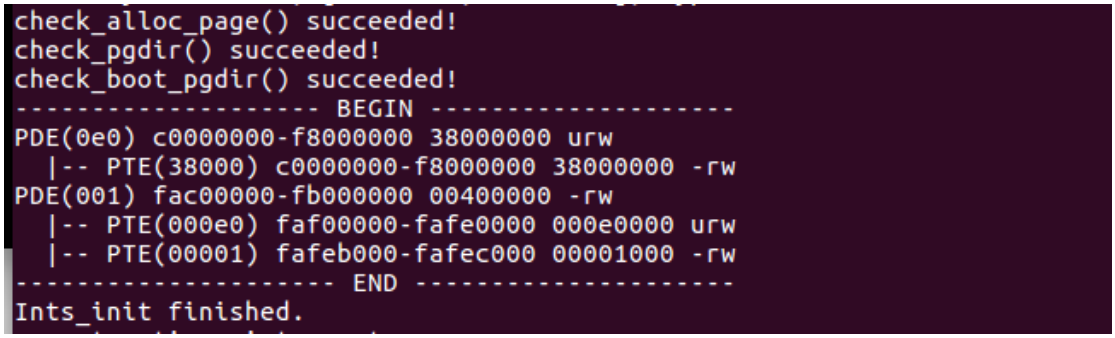
```
//page_remove - free an Page which is related linear address la and has an validated pte
void
page_remove(pde_t *pgdir, uintptr_t la) {
    pte_t *ptep = get_pte(pgdir, la, 0);
    if (ptep != NULL) {
        page_remove_pte(pgdir, la, ptep);
    }
}
```

Page_remove

```
    if ((*ptep & PTE_P) == 0) return;
    struct Page * page = pte2page(*ptep);
    if (page_ref_dec(page) == 0) free_page(page);
    tlb_invalidate(pgdir, la);
    memset(ptep, 0, sizeof(*ptep));
    //cprintf("remove_END\n");
```

Page_remove_pte

全部练习完成后，执行“**make qemu**”，部分结果如下：



3.2.4 Lab2 总结

Lab2 的主要内容是实现内存的页式管理。通过实验我了解了 OS 的页面和多级页表项是如何实现和管理的。并且熟悉了线性地址->页表项的转换。

通过实现页面的分配和释放加深了对页面引用和页面权限的了解。

3.3 Lab3 虚拟内存管理

了解虚拟内存的 Page Fault 异常处理实现，了解页替换算法在操作系统中的实现。借助于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，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。

3.3.1 给未被映射的地址映射上物理页

本练习需完场 **do_default()**函数。在验收实验时，陆老师重点考察了我们对本函数部分的理解。该函数的核心部分如下。

根据错误信息 error_code，若值为 2 或 3，且虚拟内存区域的权限不可写，则出现写异常。输出错误信息并结束。若值为 1 或 0 则代表由于读操作发生异常。

随后根据线性地址 addr，获取所在页表项。若不存在映射关系（返回链表头），则分配一块内存区域。

```
/*LAB3 EXERCISE 1: YOUR CODE*/
ptep = get_pte(mm->pgdir, addr, 1);              //(1) try to find a pte, if pte's
if (ptep == NULL) {
    cprintf("get_pte == NULL\n");
    goto failed;
}
if (*ptep == 0) {
//(2) if the phy addr isn't exist, then alloc a page & map the phy addr with logic
    if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
        cprintf("pgdir_alloc_page(mm->pgdir, addr, perm) == NULL\n");
        goto failed;
    }

}
```

上述过程即可处理未映射问题，为未被映射的地址映射上物理页。

3.3.2 补充完成基于 FIFO 的页面替换算法

在 do_default()函数中，出现异常的可能原因是所要用的页面未被加载到内存中。页面换入代码如下：

```
if(swap_init_ok) {
    struct Page *page=NULL;
    //(1) According to the mm AND addr, try to load the content of right disk pag
    //    into the memory which page managed.
    //(2) According to the mm, addr AND page, setup the map of phy addr <---> lo
    //(3) make the page swappable.
    if ((ret = swap_in(mm, addr, &page)) != 0) {
        cprintf("swap_in(mm, addr, &page) != 0");
        goto failed;
    }
    page_insert(mm->pgdir, page, addr, perm);
    swap_map_swappable(mm, addr, page, 1);
    page->pra_vaddr = addr;
}
else {
    cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
    goto failed;
}
```

由代码可见，在完成调用换入后，还要设置页面的属性为可交换，并建立虚拟地址和物理地址之间的关系。

而页面换出的实现函数为**_fifo_map_swappable()**和**_fifo_map_out_victim()**。前者根据 FIFO 思想，将最后被访问的页面加入到队列头之后。后者的功能是查询需要被换出的页面。

```c
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.
                        Loading...
    list_add_after(head, entry);

    return 0;
}
```

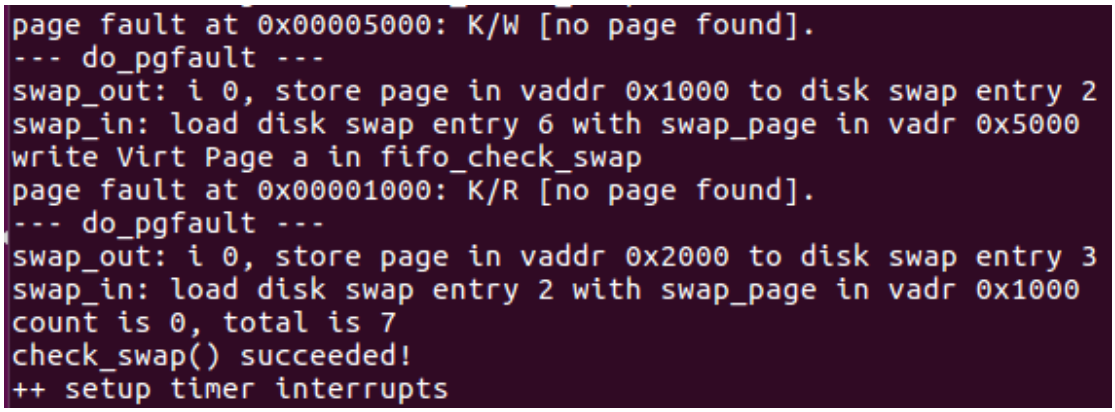
_fifo_map_swappable

```c
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 *le = head->prev;
    assert(head!=le);
    struct Page *p = le2page(le, pra_page_link);
    list_del(le);
    assert(p !=NULL);
    *ptr_page = p;
    return 0;
}
```

_fifo_swap_out_victim

部分实验输出如下：

```
-------------------- 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!
 --- check_pgfault() --
page fault at 0x00000100: K/W [no page found].
```



部分实验输出

3.3.3 Lab3 总结

本实验的主要内容是处理缺页中断中的页面换入换出。通过实验我了解了虚拟内存管理背后的实现机制，以及页式内存管理下内存与磁盘页面的交互方式。

四、 实验收获与体会

Ucore 和之前做的五个实验风格都不一样，更为系统化，实现的目标也更底层。尤其是前两个实验，由于和底层打交道比较多所以难度较大。

但是在实验的过程中，我逐渐了解了课本上之前学过的知识，比如也是页式内存管理的实现机制。这种上手写一个操作系统的方式我觉得更为贴切实际，在开发的过程中我不断遇到了错误，而在解决错误的途中我才真正学到了知识。我想这应该就是学习的意义。

Ucore 的框架我认为十分人性化，代码注释很详细，根据提示可以降低很大一部分困难。由于时间关系没能在课程结束前完成更多 core 的实验有一些遗憾，在结课后应该会继续完善之后的实验。

半学期的操作系统设计到这里结束了，这段时间内虽然大部分时间是资助实验，但是陆老师的指导让我获益匪浅，而且老师的耐心与信任让我在学习过程中更有动力。

最后，感谢陆老师在课程中的付出！

周育聪

2018.5