操作系统实验报告

实验六 教学操作系统 ucore 实验

2019-4-5

北京理工大学 计算机学院

谭超 1120161874

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

姓名/学号： 谭超 1120161874

一、 实验目的

通过对教学操作系统 ucore 的操作实验深入理解操作系统的结构与实现原理。

Lab0：

了解操作系统开发实验环境，熟悉命令行方式的编译、调试工程，掌握基于硬件模拟器的调试方式，熟悉 C 语言编程和指针的概念，了解 X86 汇编语言。

Lab1：

操作系统也是一个软件，需要通过某种机制加载并运行它，在这个实验中我们将通过 bootloader 来完成这个工作。我们需要完成一个能切换到 X86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。

Lab2:

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

二、 实验内容

Lab0：

准备实验环境，了解编程开发调试的基本工具，使用硬件模拟器 QEMU，了解处理器硬件以及了解 ucore 编程方法和通用数据结构。

Lab1：

练习 1. 1：

通过静态代码分析了解操作系统镜像文件 ucore.img 是怎么一步一步生成的以及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。

练习 1.2：

a.从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。

b.在初始化位置 0x7c00 设置实地址断点，测试断点正常。

c.从 0x7c00 开始跟踪代码运行，将单步跟踪反汇编得到的代码与 bootasm.S 和 bootblock.asm 进行比较。

d.自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

练习 1.3：

分析 bootloader 进入保护模式的过程，需要了解：

a. 为何开启 A20，以及如何开启 A20

b. 如何初始化 GDT 表

c. 如何使能和进入保护模式

练习 1.4：

分析 bootloader 加载 ELF 格式的 OS 的过程，通过阅读 bootmain.c 了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 bootloader&OS。

练习 1.5：

实现函数调用堆栈跟踪函数，完成 kdebug.c 中函数 print——stackframe 的实现。

练习 1.6：

完善中断初始化和处理。编程完善 kern/trap/trap.c 中对中断向量表进行初始化的函数 idt_init 和中断处理函数 trap。

Lab2:

练习 2.1：

实现 first-it 连续物理内存分配算法。

练习 2.2：

实现寻找虚拟地址对应的页表项。

练习 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。

三、 实验环境

VirtualBox Linux 3.13.0-24-generic

四、 程序设计与实现

Lab0:

准备实验环境，我是在 Windows 系统上通过 VirtualBox 创建 Linux 虚拟机来搭建实验环境的。具体方法就是下载并解压实验书中给出的那个镜像文件，通过 VirtualBox 新建虚拟机，制定配置该虚拟硬盘即可。装好虚拟机后便可以安装 QEMU，打开虚拟机的命令窗口，通过 sudo apt-get install qemu-system 命令安装

QEMU，提示 apt 软件库中无法找到 QEMU，通过 sudo apt update 命令更新软件库，再次执行之前的安装命令即可。

Lab1：

练习 1.1：

通过执行 make “V=”可以获知生成 ucore.img 的过程。

Makefile 中生成 ucore.img 的代码如下：

```
$(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)
```

首先创建一个 10000 字节的块，然后将 bootblock 和 kernel 拷贝过去。

ucore.img 的生成依赖于 kernel 和 bootblock，接着查看 kernel 和 bootblock 的生成代码如下：

```
$(kernel): $(KOBJS)
        @echo + ld $@
        $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
        @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
        @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call symfile,kernel)

$(call create_target,kernel)
```

生成 kernel 需要首先将 kern 目录下的所有.c 文件编译生成.o 文件。另外由上图可以看出 kernel 的生成依赖于 kernel.ld 和 kernel 对象(KOBJS).

生成 bootblock 的语句如下，bootblock 依赖于 bootfiles 和 sign。

```
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)
```

要生成 bootblock 首先要生成 bootasm.o、bootmain.o 和 sign，该过程根据在执行 make “V=”命令后可以看到：

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc
  -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootas
m.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc
  -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootma
in.c -o obj/boot/bootmain.o
```

而 sign 的生成代码如下：

```
$(call add_files_host,tools/sign.c,sign,sign)
$(call create_target_host,sign,sign)
```

根据 make “V=”命令可以查看到实际执行命令如下

```
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools
/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
```

在两个依赖项操作结束之后，执行了以下操作：

```
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0775116 s, 66.1 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000185831 s, 2.8 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
138+1 records in
138+1 records out
70775 bytes (71 kB) copied, 0.000475626 s, 149 MB/s
```

即将 obj/bootblock.o 拷贝到 obj/bootblock.out,最后适用 bin/sign 将 bootblock.out 处理成 bin/bootblock。

查看 sign.c 文件可知，硬盘主引导扇区的特征有：

- 大小为 512 字节
- 最后两个字节为 0X55AA

练习 1.2：

修改 lab1/tools/gdbinit 文件内容如下：

```
1 target remote :1234
2 set architecture i8086
```

然后在 lab1 目录下执行 make debug 命令进入 gdb。然后可以通过 i r eip 命令查看当前指令的地址：

```
(gdb) i r eip
eip            0xfff0   0xfff0
```

所以内存地址是 0xffff0，通过 x/i 0xffff0 反汇编当前指令：

```
(gdb) x/i 0xffff0
   0xffff0:     ljmp   $0xf000,$0xe05b
```

这里是一个长跳转指令，地址 0xf000:0xe05b 即是 BIOS 的存储地址。

接下来设置断点：修改 lab1/tools/gdbinit 代码如下：

```
file obj/bootblock.o
set architecture i8086
target remote :1234
break *0x7c00
continue
```

然后执行命令 make debug，执行结果如下：

```
0x000f16e6 in ?? ()ture is assumed to be i8086
Breakpoint 1 at 0x7c00: file boot/bootasm.S, line 16.

Breakpoint 1, start () at boot/bootasm.S:16
(gdb)
```

然后单步调试，并通过 x/i $pc 反汇编当前指令，并与 bootasm.S 和 bootblock.asm 比较。通过比较可以得知二者相同。部分反汇编得到的指令如下：

```
(gdb) x/i $pc
=> 0x7c00 <start>:      cli
(gdb) x/i $pc
=> 0x7c01 <start+1>:    cld
```

然后在内核代码的初始化函数设置断点，修改 la1/toos/gdbinit 文件如下：

```
file bin/kernel
set architecture i8086
target remote :1234
break kern_init
continue
```

结果如下：

```
──kern/init/init.c──┐
12      int kern_init(void) __attribute__((noreturn
));
13      void grade_backtrace(void);
14      static void lab1_switch_test(void);
15
B+>16      int  [ No Source Available ]
17      kernextern char edata[], end[];
19          memset(edata, 0, end - edata);
20
21          cons_init();                // init the
 console
22          const char *message = "(THU.CST) os is
loading ...";    cprintf("%s\n\n", message);
remo24 Thread 1──────────┘                          L
ine: ??   PC: 0xfff0kern_init ABI "GNU/Linux" is not bL
settings.

0x0000fff0 in ?? ()ture is assumed to be i8086
Breakpoint 1 at 0x100000: file kern/init/init.c, line 1
7.

Breakpoint 1, kern_init () at kern/init/init.c:17
(gdb)
```

练习 1.3：

通过分析 bootasm.S 得知 bootloader 进入保护模式的过程如下：

首先关中断和清除数据段寄存器：

```
cli                                             # Disable interrupts
cld                                             # String operations increment

# 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
```

实模式下，CPU 只能访问 1MB 的地址空间，而在保护模式下，CPU 能访问到大得多的地址空间，为了和实模式兼容添加了 A20 地址线模块，当在实模式下时，通过 A20 控制 CPU 不能访问超过 1MB 的空间。开启 A20 的代码如下：

```
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
```

初始化 GDT 的过程如下：

```
    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.
    lgdt gdtdesc
    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

    # If bootmain returns (it shouldn't), loop.
```

首先加载 GDT 表，然后通过将 cr0 寄存器 PE 位置为 1 进入保护模式，进入保护

模式之后通过长跳转更新 CS 的基地址，紧接着设置段寄存器和堆栈指针，至此 GDT 初始化完成。使能和进入保护模式是通过设置 c r 0 寄存器的 PE 位为 1 。

练习 1.4：

bootmain.c 中包含四个函数，它们分别是等待磁盘函数 waitdisk()、读取一个扇区的函数 readsect()、读取一个段的函数 readseg()和 bootloader 的入口函数 bootmain()。

bootmain()的内容如下：

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // 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);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
```

通过 bootmain()可以看出 bootloader 加载 ELF 格式的 OS 的步骤：首先将 ELF 的大小为 8 个扇区的头读入到内存地址 0x10000（ELFHDR 是宏定义的 0x10000）。然后通过 ELF 文件头的 e_magic 判断该文件是否是合法 ELF 文件。如果是合法 ELF 文件，则将 ELF 头部的程序段读入到内存中，并根据该程序段找到操作系统的入口地址。

waitdisk()、readsect()、readseg()函数定义如下：

```
/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}
```

```
/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);                         // count = 1
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

```
/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}
```

根据以上函数可以得知读取磁盘扇区的过程：首先等待磁盘就绪，然后指定扇区号，发送读取指令，等待磁盘就绪，最后读取磁盘到内存。

练习 1.5：

kdebug.c 中 print_sstackframe 函数的实现如下：

```
print_stackframe(void) {
     /* LAB1 YOUR CODE : STEP 1 */
     /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
      * (2) call read_eip() to get the value of eip. the type is (uint32_t);
      * (3) from 0 .. STACKFRAME_DEPTH
      *    (3.1) printf value of ebp, eip
      *    (3.2) (uint32_t)calling arguments [0..4] = the contents in address (unit32_t)ebp +2 [0..4]
      *    (3.3) cprintf("\n");
      *    (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
      *    (3.5) popup a calling stackframe
      *           NOTICE: the calling funciton's return addr eip  = ss:[ebp+4]
      *                   the calling funciton's ebp = ss:[ebp]
      */
        uint32_t ebp=read_ebp(),eip=read_eip();
        for(int i=0;ebp!=0&&i<STACKFRAME_DEPTH;i++){
                cprintf("ebp:0x%08x eip:0x%08x args:",ebp,eip);
                uint32_t *args=(uint32_t *)ebp+2;
                for(int j=0;j<4;j++){
                        cprintf("0x%08x ",args[j]);
                }
        cprintf("\n");
        printf_debuginfo(eip-1);
        eip=((uint32_t *)ebp)[i];
        ebp=((uint32_t *)ebp)[0];
        }
}
```

执行 make qemu 命令得到以下结果：

```
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x0000
7b84 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff
0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000
130a 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x0000
0000 0x00010094
    kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4
d08e 0xfa7502a8
    <unknow>: -- 0x00007d67 --
++ setup timer interrupts
```

输出的最后一行表示的是最先压入堆栈的一层，也即是最先使用堆栈的函数，在这里就是 bootmain.c 中的 bootmain 函数，bootloader 设置的堆栈从 0x7c00 开始，而一个栈单元是 8 字节，所有最底层栈单元 ebp 是 0x7bf8。

练习 1.6：

中断描述符表的定义如下：

```
/* Gate descriptors for interrupts and traps */
struct gatedesc {
    unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
    unsigned gd_ss : 16;            // segment selector
    unsigned gd_args : 5;            // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;            // reserved(should be zero I guess)
    unsigned gd_type : 4;            // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;                // must be 0 (system)
    unsigned gd_dpl : 2;            // descriptor(meaning new) privilege level
    unsigned gd_p : 1;                // Present
    unsigned gd_off_31_16 : 16;        // high bits of offset in segment
};
```

由上述定义可以看出中断描述符表的每个表项占 8 个字节，其中 2-3 字节是段选择子 0-1 字节和 6-7 字节是偏移量，通过段选择子找到对应的基地址，然后通过基地址和偏移量之和得到中断处理程序的入口地址。

在初始化中断向量表时，所有的中断向量有 vector.c 生成存储于 vector.S 中，向量的地址存放再 vector 数组中。顺序遍历中断向量表，并通过调用 SETGATE()对 IDT 表进行填充。最后加载 IDT。

SETGATE 的定义如下：

```
#define SETGATE(gate, istrap, sel, off, dpl) {            \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;        \
    (gate).gd_ss = (sel);                                 \
    (gate).gd_args = 0;                                   \
    (gate).gd_rsv1 = 0;                                   \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;      \
    (gate).gd_s = 0;                                      \
    (gate).gd_dpl = (dpl);                                \
    (gate).gd_p = 1;                                      \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;          \
}
```

其作用是生成一个 4 字节的中断描述表项。

idt_init 代码如下：

```
idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
     *     All ISR's entry addrs are stored in __vectors. where is uintptr_t __vectors[] ?
     *     __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
     *     (try "make" command in lab1, then you will find vector.S in kern/trap DIR)
     *     You can use  "extern uintptr_t __vectors[];" to define this extern variable which will be used later.
     * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
     *     Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup each item of IDT
     * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction.
     *     You don't know the meaning of this instruction? just google it! and check the libs/x86.h to know more.
     *     Notice: the argument of lidt is idt_pd. try to find it!
     */
      extern uintptr_t __vectors[];
      int i;
      for(i=0;i<sizeof(idt)/sizeof(struct gatedesc);i++){
            SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL);
      }
      lidt(&idt_pd);
}
```

要实现每遇到 100 次始终中断就输出一次 100 ticks,可以使用一个全局变量来记录中断次数，每当其模 100 等于 0 时便输出 100ticks。

完善 trap 的代码应在 trap_dispatch 函数中添加以下代码：

```
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in
kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        ticks++;
        if(ticks%TICK_NUM==0){
              print_ticks();
        }
        break;
```

修改之后执行 make qemu 命令得到以下结果：

```
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
kbd [097] a
kbd [000]
100 ticks
kbd [107] k
kbd [000]
100 ticks
kbd [100] d
kbd [000]
100 ticks
100 ticks
```

每隔一段时间会输出一次 100 ticks，按下某个键就会输出该键的值。

Lab2:

练习 2.0：

手动将实验 1 的代码更新到实验 2 中。

练习 2.1：

first fit 算法是从空闲分区链的头 开始查找，将找到的第一个能满足该大小要求的空闲区切分出需要的大小，剩下大小的空闲区仍保留在空闲区链中。该算法会优先使用低地址的内存区，另外，由于不断的划分，可能会留下许多难以利用的、很小的空闲区，造成内存利用率较低。

在 kern/mm/memlayout.h 文件中，我们可以查看到页面结构 Page：

```
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status of the page frame
    unsigned int property;          // the num of free block, used in first fit pm manager
    list_entry_t page_link;         // free list link
};
```

空闲区链表的头的定义如下：

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

其中 free_list 是链表的头，nr_free 是链表中空闲区的个数。

根据 default_pmm.c 中的注释，可知以下几个函数的功能：

default_init 用来初始化 free_list 和 nr_free

default_init 的实现：

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

default_init_memmap 函数的作用是用参数 addr_base 和 pagge_number 来初始化一个空闲块。初始化的流程如下：对于该块中的每个页，如果该页不是保留页，如果该页为空且不是该块的第一页，置 p->property 为 0，如果该页为空且是该块的第一页，置 p->property 为该块的页面总数。对每页设置引用个数为 0 标志位 PG_property 置为 1，表示页面有效。最后将整个块加入空闲区块链表。

default_init_memmap 的实现：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    //将基地址为base的n个连续页映射到内存
    assert(n > 0);//判断n是否大于0
    struct Page *p = base;
    for (; p != base + n; p ++) {//遍历n个页
        assert(PageReserved(p));        //检查该页是否为保留位
        p->flags = p->property = 0;     //标志位清零
        SetPageProperty(P);     //设置标志位为1
        set_page_ref(p, 0);     //清除引用此页的虚拟页的个数
        list_add_before(&free_list,&(p->page_link));//加入空闲链表

    }
    nr_free+=n;//计算空闲页总数
    base->property = n;//修改base的连续空页值为n
    //list_add_before(&free_list, &(base->page_link));
}
```

default_alloc_pages 函数的作用是在空闲区块链表中找到第一个大小满足该要求的空闲区块并重新分配大小，返回地址。整个流程如下：首先遍历空闲区块链表找到第一个符合大小的空闲块，然后置 PG_reserved=1,PG_property=0，并将该空闲区块从链表取出，如果该空闲块的大小大于我们所需要的大小，计算剩下的块的大小，并将其加入到空闲块链表中。

default_aloc_pages 实现如下：

```
static struct Page *
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){   //如果空闲区块大于需要的大小
        struct Page *p=page+n;  //找到分配之后的首页
        p->property=page->property-n;
        list_add_after(&(page->page_link),&(p->page_link));//将剩余的空闲块仍添加到空闲块链表中
        }
        list_del(&(page->page_link));//将原本的块从链表中删除
        nr_free-=n;//空闲页数量减n
        for(int i=0;i<n;i++){
            struct Page *p=page+i;
          ClearPageProperty(p);
          SetPageReserved(p);
        }
    }
    return page;
}
```

default_free_pages 函数的功能是将该块重新放入到空闲区块链表中去，同时将几个相邻的小的空闲区块整合为一个大的空闲区块。

default_free_pages 代码实现：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));

    list_entry_t *le = &free_list;
    struct Page * p;
    while((le=list_next(le)) != &free_list) {
      p = le2page(le, page_link);
      if(p>base){
        break;
      }
    }
    //list_add_before(le, base->page_link);
    for(p=base;p<base+n;p++){
      list_add_before(le, &(p->page_link));
    }
    base->flags = 0;
    set_page_ref(base, 0);
    ClearPageProperty(base);
    SetPageProperty(base);
    base->property = n;

    p = le2page(le,page_link) ;
    if( base+n == p ){
      base->property += p->property;
      p->property = 0;
    }
    le = list_prev(&(base->page_link));
    p = le2page(le, page_link);
    if(le!=&free_list && p==base-1){
      while(le!=&free_list){
        if(p->property){
          p->property += base->property;
          base->property = 0;
          break;
        }
        le = list_prev(le);
        p = le2page(le,page_link);
      }
    }

    nr_free += n;
    return ;
}
```

改进空间：在该 设计方案中，需要多次遍历链表，如果链表中含有较多的较小的空闲区碎片，则时间的消耗较大。

练习 2.2：

由 kern/mm/pmm.c 中的注释得知以下函数的功能：PDX(la)指明了虚拟内存地址 la 对应的页目录表的入口地址。KADDR(pa)根据物理地址返回对应的虚拟地址。set_page_ref(page,1)表示该页被调用一次。page2pa(page)获取该页的物理地址。alloc_page()分配一个页。Memset(void *s,char c,size_t n)将从 s 开始的 n 个字节置为 c。

线性地址组成：高 10 位是页目录索引，中间 10 位是页表索引，低 12 位是偏移量。

页目录项和页表项都是 32 位的，31-12 位是基地址，11-9 表示 AVL，第 6 位表示写入位 D，第 5 位表示访问标志 A，第 2 位表示用户/管理员位 U/S，第 1 位表示读写位 R/W，第 0 位表示存在位 P。

设计思路是：先寻找页目录表项，然后检查该页表是否在内存中，若存在，将页表引用次数加 1，获得页面物理地址，若不存在，尝试将其调入内存。然后将物理地址转为虚拟地址，设置页目录表的控制位。

```
pte_t *
get_pte(pde_t *pgdir,uintptr_t la,bool create){
    pde_t *pdep=&pgdir[PDX(la)];    //获取页表
    if(!(*pdep & PTE_P)){
        struct Page *page;
        if(!create||(page=alloc_page())==NULL){
            return NULL;
        }
        set_page_ref(page,1);   //将页表引用位置1
        uintptr_t pa=page2pa(page); //获取物理地址
        memset(KADDR(pa),0,PGSITE);
        *pdep=pa|PTE_U|PTE_W|PTE_P; //设置控制位
    }
    return &((pte_t*)KADDR(PDE_ADDR(*pdep)))[PTE(la)];//返回页表项入口时要转化为虚拟地址
}
```

get_pte 函数的实现：

如果出现访问异常，需要将要访问的页面线性地址存储在 CR2 中，再分配内存将页面调入。如果出现争用页，则根据调度算法替换某个页。

练习 2.3：

算法思路：首先检查页面是否在内存中，如果在，寻找 pte 对应的页面，将该页面的引用减 1，当页面引用次数为 0 时，释放页面，清除二次页表项并将 tlb 重新刷新。

Page_remove_pte 实现如下：

```
static inline void page_remove_pte(pde_t *pgdir,uintptr_t la,pte_t *ptep){
    if(*pte &PTE_P){
        struct Page *page=pte2page(*ptep);
        if(page_ref_dec(page)==0){  //将引用减1
            free_page(page);    //当引用为0时释放
        }
        *ptep=0;    //将目录项清零
        tlb_invalidate(pgdir,la);   //如果修改的页表是CPU正在使用的页表，使之无效
    }
}
```

执行 make qemu 指令得到的实验结果：

```
ebp:0xc0116fc8 eip:0xc0100128 args:0xc0105fdc 0xc0105fc0 0x00000f32 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:29: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
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 ---------------------
++ setup timer interrupts
100 ticks
```