操作系统第一次上机作业实验报告

成员1：

专业：计算机科学与技术

学号：1310617

姓名：刘丹

成员2：

专业：计算机科学与技术

学号：1310636

姓名：赵婧涵

一、Boot Loader

（一）理论准备：

当计算机加电，首先会把 BIOS 加载进内存执行，然后 bios 从硬盘加载 mbr，之后由 mbr 来加载操作系统或者 grab，BIOS 被加载到了960k-1MB 的地方。

在 obj/boot/boot.asm 中，可以看到00007c00 <start>:，start 符号在内存中的位置“应该是”0x7c00，即这段程序“认为”它所处的位置是内存中的0x7c00，因此为了使 mbr 程序能正确的执行，bios 会将其加载到0x7c00的位置，然后跳转到0x7c00，将控制权给它。

在 JOS 实验中，当 BIOS 找到启动的磁盘后，便将 512 字节的启动扇区的内容装载到物理内存的 0x7c00 到 0x7dff 的位置，紧接着再执行一个跳转指令将 CS 设置为 0x0000，IP 设置为 0x7c00，这样便将控制权交给了 Boot Loader 程序。

本实验中， Boot Loader 的源程序是由一个叫做的 boot.S 的 AT&T 汇编程序与一个叫做 main.c 的 C 程序组成的。这两部分分别完成两个不同的功能。

1） 其中 boot.S 主要是将处理器从实模式转换到 32 位的保护模式，这是因为只有在保护模式中我们才能访问到物理内存高于 1MB 的空间；

2） main.c 的主要作用是将内核的可执行代码从硬盘镜像中读入到内存中，具体的方式是运用 x86 专门的 I/O 指令。

通过查阅附件 lab1_1/obj/boot/boot.asm 学习 boot loader 的完整过程,并通过这个文件来跟踪整个 boot loader 的执行过程。

（二）问题与作业

问题一：

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/icommand in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

确保你能回答以下几个问题：

1） At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

2） What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

3） Where is the first instruction of the kernel?

4） How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

操作过程关键点示意：

1. Set a breakpoint at address 0x7c00.Continue execution until that breakpoint.

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/i 0x7c00
=> 0x7c00:      cli
```

```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment
```

2.Trace into bootmain()

```
void
bootmain(void)
{
    7d0a:       55                      push   %ebp
    7d0b:       89 e5                   mov    %esp,%ebp
```

```
(gdb) x/2i 0x7c40
   0x7c40:      mov    $0x7c00,%esp
   0x7c45:      call   0x7d0a
```

```
# Set up the stack pointer and call into C.
movl    $start, %esp
  7c40:       bc 00 7c 00 00          mov    $0x7c00,%esp
call bootmain
  7c45:       e8 c0 00 00 00          call   7d0a <bootmain>
```

```
(gdb) b *0x7c45
Breakpoint 3 at 0x7c45
(gdb) c
Continuing.

Breakpoint 3, 0x00007c45 in ?? ()
(gdb) si
0x00007d0a in ?? ()
(gdb) si
0x00007d0b in ?? ()
```

3.Identify the begin and end of the for loop

循环开始：

```
    for (; ph < eph; ph++)
  7d47:       39 f3                   cmp    %esi,%ebx
  7d49:       73 16                   jae    7d61 <bootmain+0x57>
            // p_pa is the load address of this segment (as well
            // as the physical address)
            readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
  7d4b:       ff 73 04                pushl  0x4(%ebx)
            goto bad;
```

```
(gdb) x/10i 0x7d47
   0x7d47:      cmp    %esi,%ebx
   0x7d49:      jae    0x7d61
   0x7d4b:      pushl  0x4(%ebx)
   0x7d4e:      add    $0x20,%ebx
   0x7d51:      pushl  -0xc(%ebx)
   0x7d54:      pushl  -0x14(%ebx)
   0x7d57:      call   0x7cd1
   0x7d5c:      add    $0xc,%esp
   0x7d5f:      jmp    0x7d47
   0x7d61:      call   *0x10018
```

循环结束：

```
    for (; ph < eph; ph++)
7d5c:       83 c4 0c                add    $0xc,%esp
7d5f:       eb e6                   jmp    7d47 <bootmain+0x3d>
            // as the physical address)
            readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d61:       ff 15 18 00 01 00       call   *0x10018
```

4.Find out what code will run when the loop is finished

```
    ((void (*)(void)) (ELFHDR->e_entry))();
7d61:       ff 15 18 00 01 00       call   *0x10018
```

```
(gdb) x/5i 0x7d5f
   0x7d5f:      jmp    0x7d47
   0x7d61:      call   *0x10018
   0x7d67:      mov    $0x8a00,%edx
   0x7d6c:      mov    $0xffff8a00,%eax
   0x7d71:      out    %ax,(%dx)
```

5.Set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

```
(gdb) b *0x7d61
Breakpoint 4 at 0x7d61
(gdb) c
Continuing.

Breakpoint 4, 0x00007d61 in ?? ()
(gdb) si
0x0010000c in ?? ()
(gdb) si
0x00100015 in ?? ()
```

```
(gdb) x/i 0x0010000c
   0x10000c:    movw   $0x1234,0x472
```

1)At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

Answer:

从 BIOS 地址为0x7c32的指令开始执行32位代码：

```
movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  7c32:       66 b8 10 00             mov    $0x10,%ax
```

导致这种改变的是在这之前开启了保护模式并从实模式跳转到了保护模式：

保护模式的开启：改变 cr0寄存器，在装载 cr0之前需要先使用指令 lgdt gdtdesc 加载段表，以正确地进行段式地址变换。

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

```
lgdt    gdtdesc
  7c1e:       0f 01 16                lgdtl  (%esi)
  7c21:       64                      fs
  7c22:       7c 0f                   jl     7c33 <protcseg+0x1>
movl    %cr0, %eax
  7c24:       20 c0                   and    %al,%al
orl     $CR0_PE_ON, %eax
  7c26:       66 83 c8 01             or     $0x1,%ax
movl    %eax, %cr0
  7c2a:       0f 22 c0                mov    %eax,%cr0
```

```
(gdb) x/6i 0x7c1e
   0x7c1e:      lgdtl  (%esi)
   0x7c21:      fs
   0x7c22:      jl     0x7c33
   0x7c24:      and    %al,%al
   0x7c26:      or     $0x1,%ax
   0x7c2a:      mov    %eax,%cr0
```

```
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
    7c32:       66 b8 10 00             mov    $0x10,%ax
  movw    %ax, %ds                # -> DS: Data Segment
    7c36:       8e d8                   mov    %eax,%ds
  movw    %ax, %es                # -> ES: Extra Segment
    7c38:       8e c0                   mov    %eax,%es
  movw    %ax, %fs                # -> FS
    7c3a:       8e e0                   mov    %eax,%fs
```

```
(gdb) b *0x7c1e
Breakpoint 2 at 0x7c1e
(gdb) c
Continuing.

Breakpoint 2, 0x00007c1e in ?? ()
(gdb) si
0x00007c23 in ?? ()
(gdb) si
0x00007c26 in ?? ()
(gdb) si
0x00007c2a in ?? ()
(gdb) si
0x00007c2d in ?? ()
(gdb) si
0x00007c32 in ?? ()
(gdb) si
0x00007c36 in ?? ()
(gdb) x/10i 0x7c2d
   0x7c2d:      ljmp   $0xb866,$0x87c32
   0x7c34:      adc    %al,(%eax)
=> 0x7c36:      mov    %eax,%ds
   0x7c38:      mov    %eax,%es
   0x7c3a:      mov    %eax,%fs
```

实模式跳转到保护模式：

```
  ljmp    $PROT_MODE_CSEG, $protcseg
```

```
ljmp    $PROT_MODE_CSEG, $protcseg
  7c2d:       ea 32 7c 08 00 66 b8    ljmp   $0xb866,$0x87c32
```

2)What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

Answer:

boot loader 执行的最后一条指令是：

```
        ((void (*)(void)) (ELFHDR->e_entry))();
    7d61:       ff 15 18 00 01 00       call   *0x10018
```

((void (*)(void)) (ELFHDR->e_entry))();　　即 call　*0x10018

加载 ELF 格式的内核镜像，并调用 ELF 的入口。

内核执行的第一条指令：

movw　$0x1234,0x472

```
(gdb) b *0x7d61
Breakpoint 4 at 0x7d61
(gdb) c
Continuing.

Breakpoint 4, 0x00007d61 in ?? ()
(gdb) si
0x0010000c in ?? ()
(gdb) si
0x00100015 in ?? ()
```

```
f010000c <entry>:
f010000c:       66 c7 05 72 04 00 00    movw   $0x1234,0x472
```

```
entry:
        movw    $0x1234,0x472                   # warm boot
```

```
(gdb) x/i 0x0010000c
   0x10000c:    movw   $0x1234,0x472
```

3)Where is the first instruction of the kernel?

Answer:由2）可知在地址0x10000c

4)How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

Answer:在 ELFHDR（即 ELF 文件格式的文件头）中有 e_phnum 成员变量表示需要载入几个程序段，每个程序段都会标识自己的大小。

```
diana@Lenovo-Ideapad:~/os/lab1_1$ objdump -f obj/kern/kernel

obj/kern/kernel:     文件格式 elf32-i386
体系结构：i386，  标志 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0010000c
```

```
diana@Lenovo-Ideapad:~/os/lab1_1$ objdump -h obj/kern/kernel

obj/kern/kernel:     文件格式 elf32-i386

节：
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001a17  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000006ec  f0101a20  00101a20  00002a20  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003895  f010210c  0010210c  0000310c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001929  f01059a1  001059a1  000069a1  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000660  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      00000029  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
```

bootmain 所做的工作就是先读到 file off 得到在文件中的偏移，然后再读取 Size 个字节，之后放到 LMA 所指定的地址处。

之后 bootmain 找出这个 ELF 文件的 entry（也在 ELF 头里），然后跳转到这个头执行。

二、The Kernel

（一）理论准备

操作系统内核经常被链接和执行在一个较高的虚拟地址，比如 0xf0100000，已将内存的较低部分留给用户程序。但是由于实际的物理内存可能不存在这个地址，我们需要通过处理器的内存管理设备将虚拟地址的 0xf0100000 到物理地址的 0x00100000，前者是链接地址，即内核假设的运行地址，后者则是内核实际的载入地址。这样一来，内核的虚拟地址可以足够以留给足够的空间给用户程序，同时也可将内核载入到 1MB 内存的顶端，即在 BIOS 的上部。事实上，我们将映射整个底部的 256MB 物理地址空间，即 0x0000000~0x0fffffff，到虚拟地址空间的 0xf0000000~0xffffffff。这也是为什么 JOS 内核被限制在只能使用256MB 物理内存。

（二）问题与作业

问题 2
1)Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?
2)Explain the following from console.c:

```
if (crt_pos >= CRT_SIZE) {
        int i;

        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
}
```

1)Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Answer:console.c 为 printf.c 提供了 cputchar(int c)接口，被 printf.c 中的 putch(int ch, int *cnt)函数调用，cputchar(int c)用于将一个字符输出到显示器，putch(int ch, int *cnt)用以输出一系列字符到控制台。

2)Explain the following from console.c:

```
if (crt_pos >= CRT_SIZE) {
        int i;

        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
}
```

代码是当 CGA 屏幕已经写满时，将最上面一行的内容抛弃，其他行的内容一次向上移动一行，为新一行的输入留出空间，即向上滚动一行。同时将光标置于最后 一行的行首。

作业 1
We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

在 printmft.c 中添加：（类比十六进制）

```
// (unsigned) octal
case 'o':
        // Replace this with your code.
        //putch('X', putdat);
        //putch('X', putdat);
        //putch('X', putdat);
        //break;
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

结果：

```
diana@Lenovo-Ideapad:~/os/lab1_1$ make qemu
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
```

可以看到输出：6828 decimal is 15254 octal!

练习 3
To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

Answer:

```
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 14                sub    $0x14,%esp
```

每次调用 call 先把当前 CS 和 IP 压栈，1个双字

```
f0100040:   55          push  %ebp          //将栈底指针 ebp 入栈，1个双字
f0100041:   89 e5       mov   %esp,%ebp
f0100043:   53          push  %ebx          //将基址寄存器 ebx 入栈，1个双字
f0100044:   83 ec 14    sub   $0x14,%esp    //将栈顶指针向低地址移动20bytes，5个
                                            //双字
```

故每次调用共4+4+4+20=32bytes，8个双字入栈。

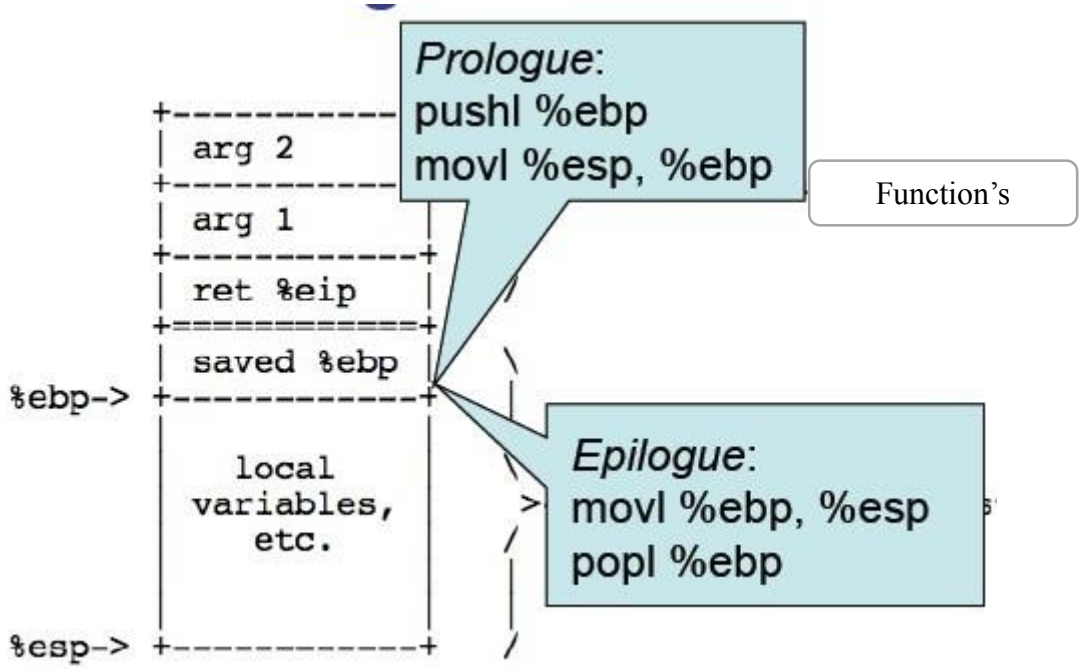
作业 2

You can do mon_backtrace()entirely in C,You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.The backtrace function should display a listing of function call frames in the following format:

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

2-1 设计思路

在内存中栈底是固定的，栈顶是变化的，JOS 中的函数调用后堆栈的结构：



esp 的含义是“这个地址以下的空间是未被使用的堆栈控件”，ebp 的含义是“这个地址以下至 esp 的空间是属于目前所执行函数的堆栈空间。

通过阅读汇编代码可以发现，一个函数在调用之前，其调用者会将参数压栈，也就是压入 arg2 和 arg1，然后调用 call，call 的动作会把 ret%eip 压栈，同时转到函数体执行，在函数体执行的开头有一段预处理代码，即图中的 prologue，会将 ebp 寄存器(call 指令不改变 ebp 的值，此时的 ebp 还是上一个函数的）内容压栈，然后将当前 esp 赋值给 ebp，随后进行现场保存的工作，存储在 local variables 空间里，值得注意的是，在预处理时会一下申请足够的空间，包括保存现场所需空间，局部变量所需空间，调用其它函数所压入变量的空间，意即图中 arg1,arg2是属于上一个函数的 local variables 空间，故 backtrace 不能准确的判断出函数所传参数个数而统一要求打印出5个参数。

因此，通过 ebp 不断寻找上层的 ebp，直到回溯所有的函数，在 entry.S 中可以看到，在调用 i386_init 之前，将 ebp 置0了，因此当 ebp 为0的时候就是函数返回的时候。

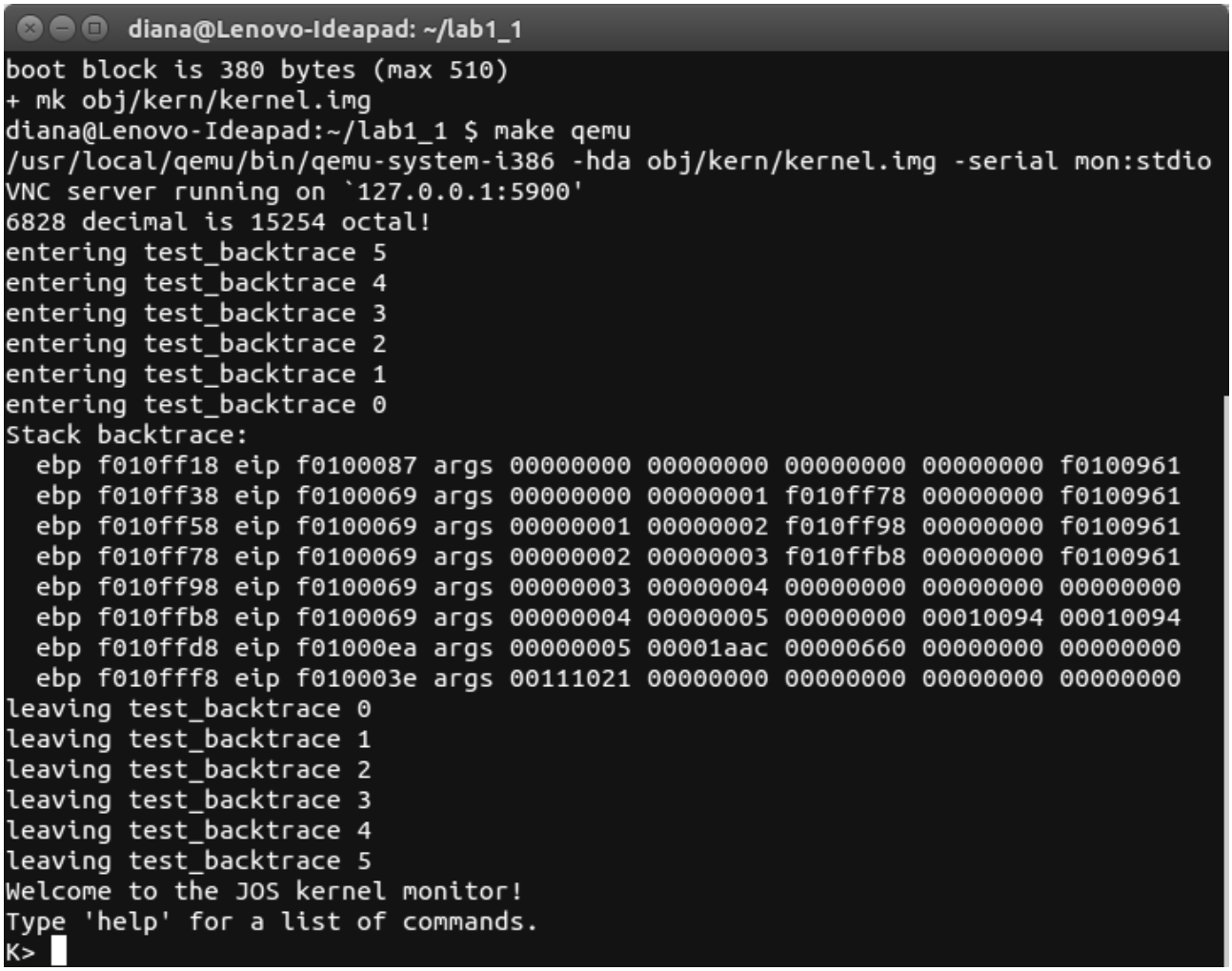
2-2 代码

kern/monitor.c 中添加代码

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.
        cprintf("Stack backtrace:\n");

        uint32_t bp,ip,arg1,arg2,arg3,arg4,arg5;
        bp=read_ebp();//读取ebp值(read_ebp()返回值为当前的 ebp 寄存器的值)
        ip=*((uint32_t*)bp+1);//从ebp指向的堆栈位置读取函数调用返回地址
        arg1=*((uint32_t*)bp+2);
        arg2=*((uint32_t*)bp+3);
        arg3=*((uint32_t*)bp+4);
        arg4=*((uint32_t*)bp+5);
        arg5=*((uint32_t*)bp+6);//从ebp指向的堆栈位置读取函数的参数
        while(bp!=0)
        {
                cprintf("  ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",bp,ip,arg1,arg2,arg3,arg4,arg5);
                bp=*(uint32_t*)bp;//读取外层函数的ebp
                if(bp!=0)
                {
                        ip=*((uint32_t*)bp+1);
                        arg1=*((uint32_t*)bp+2);
                        arg2=*((uint32_t*)bp+3);
                        arg3=*((uint32_t*)bp+4);
                        arg4=*((uint32_t*)bp+5);
                        arg5=*((uint32_t*)bp+6);
                }
        }
        return 0;
}
```

2-3 结果

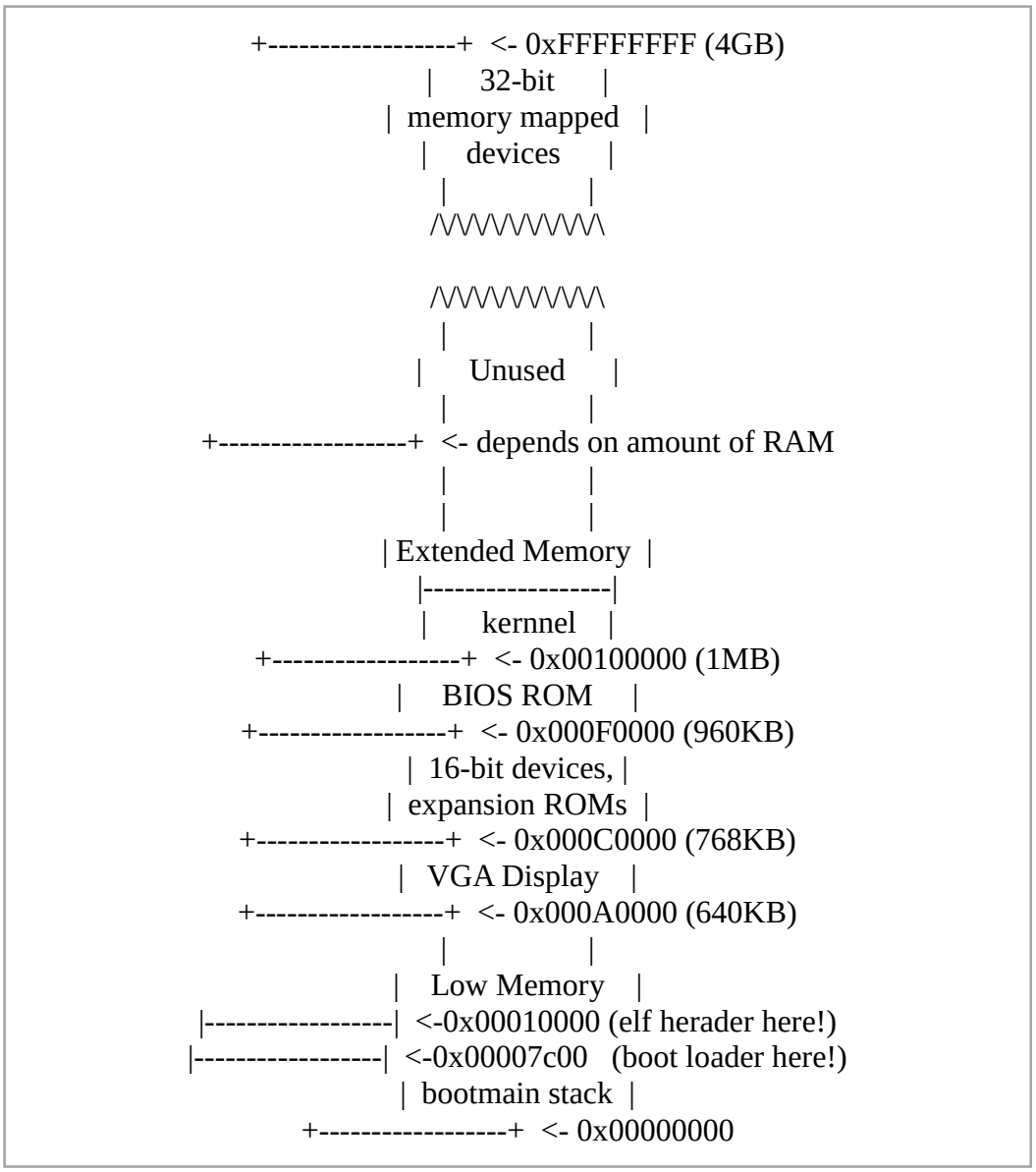

diana@Lenovo-Ideapad: ~/lab1_1
boot block is 380 bytes (max 510)
+ mk obj/kern/kernel.img
diana@Lenovo-Ideapad:~/lab1_1 $ make qemu
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f0100961
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f0100961
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f0100961
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f0100961
ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000660 00000000 00000000
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

三、内存管理

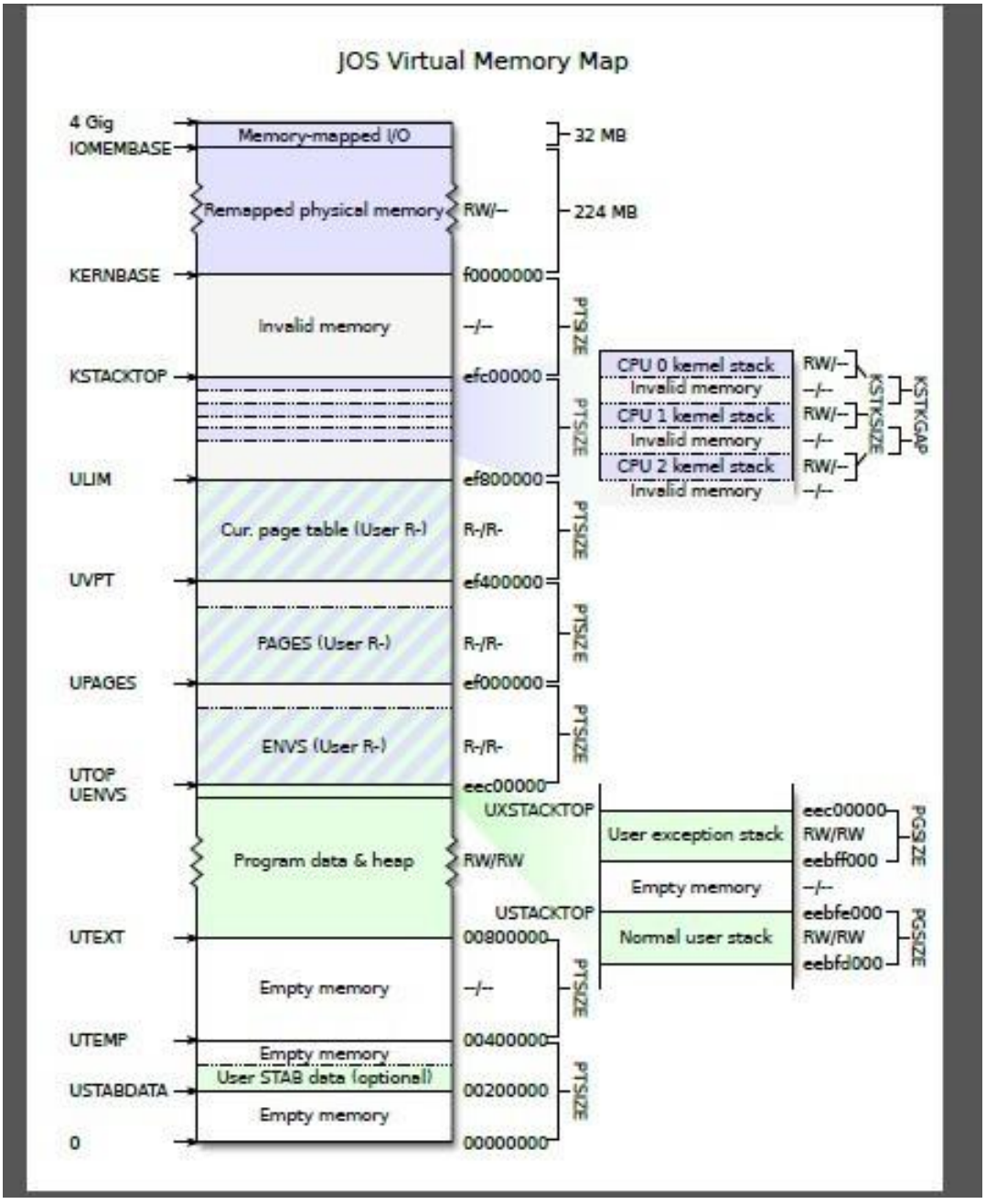
（一）理论准备

内存管理有两个组成部分。第一部分是内核的物理内存分配器，它使得内核可以分配和释放内存。分配器将以 4096 字节为单元进行操作，这被称为页。第二部分是虚拟内存，在此我们将初步建立起内存管理单元(MMU,Memory Management Unit),其作用是将软件所使用的虚拟内存地址映射为硬件内存中的实际物理地址。

物理内存的分布：



虚拟内存分布：



物理页管理：除了要设置处理器的硬件来将虚拟地址正确的转换为物理地址,操作系统也必须要记录每一部分的物理 RAM 是空闲还是被使用的。 JOS 将会使用页粒度(page granularity)来管理物理内存,这样就可以使用 MMU 来映射和保护每一份被分配的内存。

虚拟地址、线性地址和物理地址：在 x86 术语中，一个虚拟地址包括一个段选择器和一个段中的 offset，一个线性地址是你在段转换之后、页转换之前获得的。一个物理地址是

你在段、页转换后最终获得的、最终出现在硬件总线中。虚拟地址的的 offset 是一个 C 指针。在 boot/boot.S 中，我们安装一个 Descriptor Table(GDT)，它通过将所有段的基地址设为 0，并限制在 0xffffffff 以内，可以高效的禁用段转换。由此，选择器没有作用，线性地址经常等于虚拟地址的 offset。在本实验中，我们扩充这个为：映射物理内存的前256MB，从虚拟地址 0xf0000000 开始，同时映射到虚拟内存的其他区域。

（二）问题与作业

作业 3
在文件 kern/pmap.c 中,你需要实现以下函数的代码(如下,按序给出):
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
check_page_free_list() 和 check_page_alloc()将测试你的物理页分配器。你需要引导 JOS 然后查看 check_page_alloc()的成功报告。加入你自己的 assert()来验证你的假设是否正确将会有所帮助。

3-1 原理：

JOS 使用 Page 数据结构来管理内存，一个 Page 代表一个 PGSIZE（4K）大小的物理页面，Page 数据结构的定义在 memlayout.h 中，共有两个域，第一个域是 pp_link，指向下一个空闲 Page 结构的指针，第二个域是一个 short 整形，代表当前此物理页 面的引用次数，若为0则是没有被引用也就是空闲页面。

使用 page_free_list 维护一个空闲物理内存的链表，page_free_list 本身就是一个 Page 指针，然后通过 Page 结构体里面的 pp_link 域构成空闲链表。page_free_list 链表是从 pages 数组的末尾开始从高地址指向低地址。

一个 Page 代表4K，在 pmap.c 中的 i386_memory_detect 函数中检测内存后，使用总物理内存/4k 得到所需要的 Page 数量，赋值给 npages，即 npages 代表所需 Page 结构体的数量。

所有 Page 在内存（物理内存）中的存放是连续的，存放于 pages 处，可以通过数组的形式访问各个 Page，而 pages 紧接于 end[]符 号之上，end 符号是编译器导出符号，其值约为 kernel 的 bss 段在内存（虚拟内存）中的地址+bss 段的段长，对应物理和虚拟内存布局也就是在 kernel 向上的紧接着的高地址部分连续分布着 pages 数组。

JOS 提供 page2pa, pa2page 等函数可以进行 Page 数据结构的指针向物理地址的转换，或反转换等。

3-2 函数实现（见附件 lab1_2/kern/pmap.c）

3-2-1boot_alloc()

所做工作：在初始化时，free_page_list 还未建立时用到：开辟出 n 字节的空闲空间，并返回其首地址。

设计思路：end 符号向上均为未使用空间，只要返回这些空间就行。

首先将 end 符号向上和4K 字节对齐（JOS 已经帮我们完成），然后将 nextfree 加上要分配的空间并依然4K 字节对齐，接着返回原先的 nextfree 即可。

代码：

```
static void *
boot_alloc(uint32_t n)//end符号向上均为未使用空间，只要返回这些空间就行
{
        static char *nextfree;  // virtual address of next byte of free memory
        char *result;

        // Initialize nextfree if this is the first time.
        // 'end' is a magic symbol automatically generated by the linker,
        // which points to the end of the kernel's bss segment:
        // the first virtual address that the linker did *not* assign
        // to any kernel code or global variables.
        //end是链接器作链接时得到的内核结束地址
        if (!nextfree)
        {//首先将end符号向上和4K字节对齐
                extern char end[];
                nextfree = ROUNDUP((char *) end, PGSIZE);
        }

        // Allocate a chunk large enough to hold 'n' bytes, then update
        // nextfree.  Make sure nextfree is kept aligned
        // to a multiple of PGSIZE.
        //
        // LAB 2: Your code here.

        //然后将这个地址（也就是nextfree）加上要分配的空间并依然4K字节对齐，接着返回原先的nextfree
        result=nextfree;
        nextfree+=ROUNDUP(n,PGSIZE);
        return result;

}
```

3-2-2mem_init()

所做工作：为 pages 分配内存

代码：

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct Page's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct Page in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:
pages=(struct Page*)boot_alloc(npages*sizeof(struct Page)); //给pages分配空间
```

3-2-3page_init()

所做工作：pages 数组以及 page_free_list 的初始化

设计思路：

在 page_init()里系统首先初始化了 pages 数组以及 page_free_list，可以看到这个 page_free_list 指向了所有的 Page 结构，换句话说此时认为所有的页面都是空闲可分配的，我们需要做的是要从中把一些我们已经用的内存页面从中剔除出去，这包括0地址向上的第一个页面（包括 IDT 等），IO hole（0xA0000--0x100000，包括 vga display ,bios 等），kernel 地址之上的部分（kernel 本身+kern_pgdir+pages）。巧合的是，IO hole，和 kernel 之上部分是连续的地址，因为 kernel 就加载在0x100000处，所以其实只需要剔除两块地址，第一块是0地址开始的第一个 页面，第二块就是 io hole 开始的向上的一组连续的页面。

page_free_list 链表是从 pages 数组的末尾开始从高地址指向低地址，所以先计算出要剔除的 Page 的地址，然后通过指针操作剔除即可。

代码：

```
//剔除两块地址

//第一块是0地址开始的第一个页面
extern char end[];
pages[1].pp_link=0;//只需让第二个页面（下标为1）的pp_link域指向空，其原本其指向的是第一个页面


//第二块是io hole开始的向上的一组连续的页面

//先计算出要剔除的Page的地址
struct Page* pgstart=pa2page((physaddr_t)IOPHYSMEM);//首地址
struct Page* pgend=pa2page((physaddr_t)(end-KERNBASE+PGSIZE+npages*sizeof(struct Page)));

//pgstart和pgend这两个Page也要剔除
pgend++;
pgstart--;

pgend->pp_link=pgstart;//改变pp_link域，跳过中间的区域（要剔除的部分）
```

3-2-4page_alloc()

所做工作：页面分配

设计思路：从 page_free_list 头剔除一个 Page，然后改变 page_free_list 使其为 pp_link。

代码：

```
struct Page *
page_alloc(int alloc_flags)//从page_free_list头剔除一个Page，然后改变page_free_list使其为其pp_link
{
        // Fill this function in
        if(!page_free_list)
                return NULL;
        struct Page* pp=page_free_list;
        page_free_list=page_free_list->pp_link;
        if(alloc_flags& ALLOC_ZERO)
                memset(page2kva(pp),0,PGSIZE);
        return pp;

}
```

3-2-5page_free()

所做工作：页面释放

设计思路：将释放页面的 pp_link 指向 page_free_list 头，然后改变 page_free_list 使其为 pp（要释放页面）。

代码：

```
// Return a page to the free list.
// (This function should only be called when pp->pp_ref reaches 0.)
//
void
page_free(struct Page *pp)
{
        // Fill this function in

        assert(pp->pp_ref == 0 || pp->pp_link == NULL);
        pp->pp_link=page_free_list;
        page_free_list=pp;
}
```

作业3检测结果：

```
diana@Lenovo-Ideapad:~/os/lab1_2$ make qemu
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:705: assertion failed: page_insert(kern_pgdir, pp1,
0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

问题 3

假设以下内核代码是正确的,那么变量 x 将会是什么类型,uintptr_t 或者 physaddr_t?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

Answer：value 是可以直接操作虚拟内存的 char * ，x 是从 value 强制转化而来，没有经过虚拟内存转物理内存的步骤，所以 mystery_t 应该也是虚拟内存的一种，故 x 是 uintptr_t 类型。

作业 4

在文件 kern/pmap.c 文件中,你必须实现以下函数的代码。

pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()

mem_init()调用的 check_page(),用于测试你的页表管理方法。

4-1 原理

首先从硬件机制说起，当 cpu 拿到一个地址并根据这个地址访问主存时，在 x86体系架构下要经过至少两级的地址变换，第一级成为段式地址变换而第二级成为页式地址变换。

最原始的地址叫做虚拟地址，根据规定，将前16位作为段选择子，后32位作为偏移。根据段选择子查找 gdt/ldt，查到的内容替加上偏移，此时的地址就变成了线性地址。

线性地址前10位被称作页目录入口（page directory entry 也就是 pde），其含义为该地址在页目录中的索引，中间10位为页表入口（page table entry，也就是 pte），代表在页表中的索引，最后12位是偏移。

当一个线性地址进入页式地址变换机制时，首先 cpu 从 cr3寄存器里得到页目录（page directory）在主存中的地址，然后根据这个地址加上 pde 得到该地址在页目录中对应的项。无论是页目录的项还是页表的项均是32位，前20位为地址，后12位为标志位。当获取了相应的页目录项之后，根据前20位地址得到页表所在地址，加上偏移 pte 得到页表项，取出前20位加上线性地址本身的后12位组成物理地址，整个变换过程结束。

这个过程完全是由硬件实现，在这个部分的实验中要做的是初始化并维护页目录与页表，当页目录与页表维护好了，然后使 cr3装载新的页目录，一切就交由硬件去处理地址变换了。

4-1 函数实现（见附件 lab1_2/kern/pmap.c）

4-2-1 pgdir_walk()

所做工作：用于查找某个虚拟地址是否有页表项，如果没有也可以通过此函数创建，值得注意的是，有页表项并不代表已经被映射。

设计思路：查找页目录表，根据宏 PDX 取得页目录项（相关宏定义在 mmu.h 中），如果不为空，取出该项内容的前20位（PTE_ADDR 宏），这是物理地址，通过此物理地址查找对应的 Page 结构（pa2page 宏），然后获得此 Page 的虚拟地址（page2kva 宏）。

此时的地址为页表的虚拟地址，根据偏移得到页目录项，在返回此页目录项地址。

如果前20位不为空，检查 create，如果为0，返回 null。否则新分配一个 Page 作为页表，然后自增 Page 的引用，让该页目录项的前20位为页表物理地址（page2pa 得到物理地址），并设置一些权限符号（不加通不过最后的检测函数），在通过此页表的虚拟地址得到相应页表项的虚拟地址并返回。

代码：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{//用于查找某个虚拟地址是否有页表项，如果没有也可以通过此函数创建，值得注意的是，有页表项并不代表已经被映射。
        // Fill this function in
 //pgdir 本身是虚拟地址 解引用得到的是page directory的物理地址
 //关键物理地址和虚拟地址之间的转化
    pte_t* result=NULL;
    if(pgdir[PDX(va)]==(pte_t)NULL)
    {
        if(create==0)
            return NULL;
        else
        {
            struct Page* page=page_alloc(1);
            if(page==NULL)
                return NULL;
            page->pp_ref++;
            pgdir[PDX(va)]=page2pa(page)|PTE_P|PTE_W|PTE_U;
            result=page2kva(page);
        }
    }
    else
        result=page2kva(pa2page(PTE_ADDR(pgdir[PDX(va)])));
    return &result[PTX(va)];
}
```

4-2-2boot_map_region()

所做工作：该函数把虚拟地址[va,va+size)的区域映射到物理地址 pa 开始的内存中去。

设计思路：函数中利用 pgdir_walk()解引用得到物理地址。

代码：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{//把虚拟地址[va,va+size)的区域映射到物理地址pa开始的内存中
        // Fill this function in
    uintptr_t  va_next = va;//虚拟地址
    physaddr_t pa_next = pa;//物理地址
    pte_t* pte = NULL;//页表入口

    ROUNDUP(size,PGSIZE);//对齐

    assert(size%PGSIZE == 0 || cprintf("size:%x \n",size));

    int temp = 0;
    for(temp = 0;temp < size/PGSIZE;temp++)
    {
        pte = pgdir_walk(pgdir,(void *)va_next,1);

        if(!pte)
            return;

        *pte = PTE_ADDR(pa_next) | perm | PTE_P;
        pa_next += PGSIZE;
        va_next += PGSIZE;
    }
}
```

4-2-3page_lookup()

所做工作：page_lookup 函数检测 va 虚拟地址的虚拟页是否存在。

设计思路：不存在返回 NULL，存在返回描述该虚拟地址关联物理内存页的描述结构体 PageInfo 的指针(PageInfo 结构体仅用来描述物理内存页)。

代码：

```
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)//检测va虚拟地址的虚拟页是否存在
{
        // Fill this function in
    // Return NULL if there is no page mapped at va.
    pte_t* pte=pgdir_walk(pgdir,va,0);// 使用pgdir_walk查找pte
    if(!pte) //如果为空则说明没有映射，返回NULL
        return NULL;
    if(pte_store!=0)  //根据pte_store是否为0，先把此pte的地址存到pte_store里
        *pte_store=pte;

    if(pte[0] !=(pte_t)NULL)  //页表项是否为0
        return pa2page(PTE_ADDR(pte[0])); //为0，说明地址没有被映射（有页表项不代表被映射），返回NULL
    else  //不为0，返回这个页表项的前20位所组成的物理地址所对应的Page结构
        return NULL;
}

//
```

4-2-4page_remove()

所做工作：清除 va 所在虚拟内存页。

设计思路：把 va 关联物理页的 page table entrance 置为 NULL（ page table entrance 解引用得到的就是物理页地址)。

代码：

```
void
page_remove(pde_t *pgdir, void *va)//清除va所在虚拟内存页
{//把va关联物理页的page table entrance（可由pgdir_walk(pgdir,va,0)找到）置为NULL即可
        // Fill this function in
    pte_t*  pte = pgdir_walk(pgdir,va,0);
    pte_t** pte_store = &pte;
    struct Page* pp = page_lookup(pgdir,va,pte_store);

    if(!pp)
        return ;

    page_decref(pp);
    **pte_store = 0;
    tlb_invalidate(pgdir,va);
}
```

4-2-5page_insert()

所做工作：把 pp 描述的物理页与虚拟地址 va 关联起来。

设计思路：假设 va 所在的虚拟内存页不存在，那么 pgdir_walk 的 create 为1,创建这个虚拟页；假设 va 所在的虚拟内存页存在，那么取消当前 va 的虚拟内存页也和之前物理页的关联，而且为 va 建立新的物理页联系——pp 所描写叙述的物理页。

代码：

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)//把pp描述的物理页与虚拟地址va关联起来
{
        // Fill this function in
    pte_t* pte = pgdir_walk(pgdir,va,0);
    physaddr_t ppa = page2pa(pp);

    if(pte)
    {
        if(*pte & PTE_P)
            page_remove(pgdir,va);//取消va与之物理页之间的关联
        if(page_free_list == pp)
            page_free_list = page_free_list->pp_link;//更新链表头
    }
    else//va所在的虚拟内存页不存在
    {
        pte = pgdir_walk(pgdir,va,1);//创建虚拟页
        if(!pte)
            return -E_NO_MEM;
    }

    *pte = page2pa(pp) | PTE_P | perm; //建立va与pp描述物理页的联系

    pp->pp_ref++;
    tlb_invalidate(pgdir,va);
    return 0;
}
```

作业4检测结果：

```
diana@Lenovo-Ideapad:~/os/lab1_2$ make qemu
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
VNC server running on `127.0.0.1:5901'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic at kern/pmap.c:707: assertion failed: check_va2pa(pgdir, UPAGES + i
) == PADDR(pages) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

作业 5

在调用 check_page()之后,填写 mem_init()丢失的代码。

你的代码需要通过 check_kern_pgdir()和 check_installed_pgdir 的检验。

思路：利用 boot_map_region()映射相应区域即可。

代码（见附件 lab1_2/kern/pmap.c）：

UPAGES 的映射

```
//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
// Your code goes here:
//UPAGES的映射
boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages*sizeof(struct Page), PGSIZE), PADDR(pages), PTE_U|PTE_P);
```

内核栈的映射

```
// use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:
//内核栈的映射
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, ROUNDUP(KSTKSIZE,PGSIZE), PADDR(bootstack), PTE_W|PTE_P);

//////////////////////////////////////////////////////////////////////
```

整块内存的映射

```
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
//整块内存的映射
boot_map_region(kern_pgdir, KERNBASE, ROUNDUP(~KERNBASE, PGSIZE), 0, PTE_W|PTE_P);
```

作业5检验结果：

```
diana@Lenovo-Ideapad:~/os/lab1_2$ make qemu
/usr/local/qemu/bin/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio
VNC server running on `127.0.0.1:5901'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

问题 4

1)在这一点上页目录中的哪些行已经被填写了?他们映射了什么地址,指向了哪?换言之,尽量填写以下表:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

2)我们已经将内核和用户环境放在了相同的地址空间。为什么用户程序不能读或者写内核内存?什么样的具体机制保护内核内存?

3)这个操作系统最大能支持多大的物理内存?为什么?

4)管理内存有多大的空间开销,如果我们拥有最大的物理内存?这个空间开销如何减小?

Answer:

1) 在这一点上页目录中的哪些行已经被填写了?他们映射了什么地址,指向了哪?

Entry	**Base Virtual Address**	**Points to(logically)**
1023	0xffc0000	page table for top 4MB of phys memory
1022	0xff80000	page table for 248MB--(252MB-1) phys mem
.	.	page table for ... phys mem
960	0xf000000(KERNBASE)	page table for kernel code & static data 0--(4MB-1) phys mem
959	0xefc0000(VPT)	page directory self(kernel RW)

958	0xef80000(ULIM)	page table for kernel stack
957	0xef40000(UVPT)	same as 959 (user kernel R)
956	0xef00000(UPAGES)	page table for struct Pages[]
.	.	NULL
2	0x00800000	NULL
1	0x00400000	NULL
0	0x00000000	same as 960 (then turn to NULL)

2)我们已经将内核和用户环境放在了相同的地址空间。为什么用户程序不能读或者写内核内存?什么样的具体机制保护内核内存?

页表和页目录有 PTE_U 位，可以来控制用户是否可以访问某页。

2) 这个操作系统最大能支持多大的物理内存?为什么?

在 lab1_2/inc/mmu.h 中，可以看到如下内容：

```
// Page directory and page table constants.
#define NPDENTRIES      1024            // page directory entries per page directory
#define NPTENTRIES      1024            // page table entries per page table

#define PGSIZE          4096            // bytes mapped by a page
#define PGSHIFT         12              // log2(PGSIZE)

#define PTSIZE          (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT         22              // log2(PTSIZE)

#define PTXSHIFT        12              // offset of PTX in a linear address
#define PDXSHIFT        22              // offset of PDX in a linear address
```

在 lab1_2/inc/memlayout.h 中可以看到如下内容：

```
// User read-only virtual page table (see 'vpt' below)
#define UVPT            (ULIM - PTSIZE)
// Read-only copies of the Page structures
#define UPAGES          (UVPT - PTSIZE)
// Read-only copies of the global env structures
#define UENVS           (UPAGES - PTSIZE)
```

```
struct Page {
        // Next page on the free list.
        struct Page *pp_link;

        // pp_ref is the count of pointers (usually in page table entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count fields.

        uint16_t pp_ref;
};
```

最大能支持2GB 物理内存。原因： UPAGES 最大是 4MB, sizeof(struct PageInfo))=8Bytes, 所以最多有4MB/8B=512K 页, 一页的大小是 4KB,故最大能支持4MB/8B*4KB=2GB 物理内存。

3) 管理内存有多大的空间开销,如果我们拥有最大的物理内存?这个空间开销如何减小?

在 lab1_2/inc/mmu.h 中可以看到如下内容：

```
// Page directory and page table constants.
#define NPDENTRIES      1024            // page directory entries per page directory
#define NPTENTRIES      1024            // page table entries per page table

#define PGSIZE          4096            // bytes mapped by a page
#define PGSHIFT         12              // log2(PGSIZE)

#define PTSIZE          (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT         22              // log2(PTSIZE)

#define PTXSHIFT        12              // offset of PTX in a linear address
#define PDXSHIFT        22              // offset of PDX in a linear address
```

PageInfo 开销 ：4MB（一个 PageInfo Object 代表一个物理页）

页表开销：2MB，一个页表有1K 个页表项,每个页表项是4Bytes，即每个页表是4KB（一页的大小）,共有页表项512K 项（对应512K 物理页），需要$512K/1K = 512$个页表，故空间开销是$512 \times 4\text{KB} = 2\text{MB}$;

页目录开销：4KB，一个页目录项对应一个页表，512个页表只需一个页目录表即可，一个页目录有1K 个页目录项, 每个页目录项是4Bytes，即每个页目录是4KB（一页的大小），故空间开销是4KB;

空间总开销:6MB+4KB

减小开销的方法：使用多级页表

四、组员分工

第一周：Ubuntu 系统安装，环境搭建，熟悉环境并阅读相关文件。（两人）

第二、三周：两人各自完成 lab1和 lab2实验的完整过程，遇到问题一起探讨解决，各自完成报告的草本。

第四周：两人相互交流，完善改进代码，总结实验收获，完成实验最终报告。