華中科技大學

课 程 实 验 报 告

课程名称：操作系统原理课程设计——JOS

院　　系 ： 计算机科学与技术

专业班级 ： 计卓 1401

学　　号 ： U201410081

姓　　名 ： 李冠宇

指导教师 ： 张杰

2017 年 3 月 15 日

目 录

1 Booting a PC

1.1 PC Bootstrap

1.1.1 Getting Started with x86 assembly

> **Exercise 1.** Read or at least carefully scan the entire PC Assembly Language book, except that you should skip all sections after 1.3.5 in chapter 1, which talk about features of the NASM assembler that do not apply directly to the GNU assembler. You may also skip chapters 5 and 6, and all sections under 7.2, which deal with processor and language features we won't use in 6.828.
>
> Also read the section "The Syntax" in Brennan's Guide to Inline Assembly to familiarize yourself with the most important features of GNU assembler syntax.

大二学习汇编时学习的就是 Intel 的 x86 汇编，当时也研究过 C 的内联汇编的语法，所以这个 Exercise 基本直接过了。

1.1.2 Simulating the x86

> **Exercise 2.** Scan through the Using Bochs internal debugger section of the Bochs user manual to get a feel for these commands and their syntax. Play with the commands a little: do some stepping and tracing through the code, examining CPU registers and memory and disassembling instructions at different points, without worrying too much yet about what the code is actually doing. While the kernel monitor is waiting for user input (or at any other time the simulation is running), you can always hit CTRL-C in the shell window from which you ran Bochs in order to halt the simulation and break back into the Bochs debugger. Be sure you understand the distinction between which software you're interacting with when you type commands in the kernel monitor versus in the Bochs debugger.

这个 Exercise 主要是学习 Bochs 的使用方法及自带指令，常用的指令用“info”、“c”、“s”等等，语法跟命令格式与 GDB 还是挺相似的，基本都尝试了一遍后便继续前进的步伐了。

1.1.3 The PC's Physical Address Space

注意到，BIOS ROM（下一小要考虑的内容）是放在了 960KB 到 1MB 之间的内存空间中。这一部分没有 Exercise。

1.1.4 The ROM BIOS

Exercise 3. Use the Bochs debugger to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at the Bochs I/O address assignments, Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

Bochs 运行后，可以发现 CPU 加电后从地址 0xffffff0（f000:fff0）开始运行，这个地址指向 BIOS 的 ROM 部分。这里需要注意的是，CPU 刚启动时处于实模式，物理地址的计算应为 16 * 0xf000 + 0xfff0 = 0xffff0。

BIOS 先设置中断表，然后检测和初始化关键设备，然后把硬盘的第一个块加载到 0x00007c00 运行。

1.2 The Boot Loader

Exercise 4. Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. 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 `u` command in Bochs to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the GNU disassembly in `obj/boot/boot.asm` and the Bochs disassembly from the `u` command.

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.

Be able to answer the following questions:

- At exactly what point does the processor transition from executing 16-bit code to executing 32-bit code?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- 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?

1. 处理器从 BIOS 进入 boot loader 后，在 boot/boot.S 中第 48 行到第 51 行代码，如图 1.1 所示，boot loader 将寄存器 cr0 的末位更改为 1，使得处理器从实模式更改到保护模式。

```
44   # Switch from real to protected mode, using a bootstrap GDT
45   # and segment translation that makes virtual addresses
46   # identical to their physical addresses, so that the
47   # effective memory map does not change during the switch.
48   lgdt    gdtdesc
49   movl    %cr0, %eax
50   orl     $CR0_PE_ON, %eax
51   movl    %eax, %cr0
```

图 1.1 boot loader 模式切换代码

2. boot loader 执行的最后一条指令为将内核 ELF 文件载入内存后，调用内核入口点，在 boot/main.c 中的第 58 行，代码为“((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();”。 根据查询 objdump -x obj/kern/kernel 的结果可以得知内核 ELF 的入口地址为 0xf010000c， 但是 boot/main.c 在载入内核时做了一次手动的地址转换，将高位的 f 去掉了，所以事实上在运行中内核是被加载到了 0x10000c 的内存地址上，所以启动 Bochs 在 0x10000c 设下断点，这时 0x10000c 的代码 movw $0x1234，0x472 就是内核的第一条语句。这个时候我们反过头来去追溯内核 kernel 的源代码，果然 kern/entry.S 第 44 行正好就是我们找到的这一语句。
3. boot loader 从内核 ELF 文件的文件头中可以知道该 ELF 文件被分成了多少 section 和多少 program，就可以知道相应的读取数目了。这些信息可以通过 objdump -x obj/kern/kernel 得到，如图 1.2 所示。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab1]
└──╼ $ objdump -x obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xf010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0xf0100000 align 2**12
         filesz 0x00006618 memsz 0x00006618 flags r-x
    LOAD off    0x00008000 vaddr 0xf0107000 paddr 0xf0107000 align 2**12
         filesz 0x00008320 memsz 0x00008980 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

图 1.2 kernel 的 ELF 文件头

1.2.1 Loading the Kernel

Exercise 5. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 8 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R.

There are other references on pointers in C, though not as strongly recommended. A tutorial by Ted Jensen that cites K&R heavily is available in the course readings.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

大一学 C 语言和大三学 C++的时候对指针的学习还是比较透彻的，所以这个 Exerci 直接过掉了。

> **Exercise 6.** Reset the machine (exit bochs and start it again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use Bochs to answer this question. Just think.)

启动 Bochs，分别在 0x7c00（刚进入 bootloader）和 0x10000c（刚进入 kernel）处设置断点，在两个断点处分别打印 0x100000 处的 8 个 word，调试结果如图 1.3 所示。

内存 0x00100000 是内核的载入地址，内核由 Boot loader 负责载入。初始当 BIOS 切换到 boot loader 时，它还没有开始相应的装载工作，所以这个时候看所有的 8 个 word 全是 0。而当 boot loader 进入内核运行时，这个时候内核已经装载完毕，所以从 0x00100000 开始就是内核 ELF 文件的文件内容了。

```
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> b 0x7c00
<bochs:2> b 0x10000c
<bochs:3> c
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=1631781009
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): cli                       ; fa
<bochs:4> x/8x 0x100000
[bochs]:
0x00100000 <bogus+       0>:    0x00000000      0x00000000      0x00000000      0x00000000
0x00100010 <bogus+      16>:    0x00000000      0x00000000      0x00000000      0x00000000
<bochs:5> c
(0) Breakpoint 2, 0x0010000c in ?? ()
Next at t=1631804261
(0) [0x000000000010000c] 0008:0010000c (unk. ctxt): mov word ptr ds:0x472, 0x1234 ; 66c705720400003412
<bochs:6> x/8x 0x100000
[bochs]:
0x00100000 <bogus+       0>:    0x1badb002      0x00000003      0xe4524ffb      0x7205c766
0x00100010 <bogus+      16>:    0x34000004      0x15010f12      0x0010f018      0x000010b8
<bochs:7>
```

图 1.3 Exercise 6 调试结果

1.2.2 Link vs. Load Address

> **Exercise 7.** Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `gmake clean`, recompile lab1 with `gmake`, and trace into the boot loader again to see what happens. Don't forget to change the link address back afterwards!

链接地址实际上就是程序自己假设在内存中存放的位置，即编译器在编译的时候会认定程序将会连续的存放在从起始处的链接地址开始的内存空间；加载地址则

是可执行程序在物理内存中真正存放的位置。搞清楚这两个概念，这个 Exercise 基本可以通过了，关于这两个概念的区别，在本报告的后面还会继续展看讲述。

1.3 The Kernel

1.3.1 Using segmentation to work around position dependence

> **Exercise 8.** Use Bochs to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.
>
> What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in `kern/entry.S`, trace into it in Bochs, and see if you were right.

boot loader 在进行初始化数据的时候自己定义了 GDT，切换到内核运行后，内核在载入初期马上重新定义了自己的 GDT，然后替换掉了原有的 GDT。新的 GDT 和原来的不同之处在于，新的段表其基址全部变成了-KERNBASE 而不是原来的 0。每次相对寻址时，-KERNBASE 如果和内核的链接地址 0xf01xxxxx 相加的话，就自动将最高的 f 去掉。

1.3.2 Formatted Printing to the Console

> **Exercise 9.** 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.

实现这个 Exercise 还是挺简单的，只需要看懂 lib/printmft.c 中十进制和十六进制打印的相关代码，八进制打印的代码很容易就可以写出来，代码如图 1.4 所示。首先利用 lib/printfmt.c 中的 getuint 函数从参数列表中取出一个无符号整数，然后设置 base 变量为 8，跳转到专门打印数字的 number 处执行即可。

```
207         // (unsigned) octal
208         case 'o':
209             num = getuint(&ap, lflag);
210             base = 8;
211             goto number;
```

图 1.4 Exercise 9 的实现代码

Be able to answer the following questions:

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`:

```
1      if (crt_pos >= CRT_SIZE) {
2              int i;
3              memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4              for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                      crt_buf[i] = 0x0700 | ' ';
6              crt_pos -= CRT_COLS;
7      }
```

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

   Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

   - In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?
   - List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

   What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise.

   The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

   Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

1. kern/console.c 与 kern/print.c 之间的交互接口是“函数调用”。具体来说，是 kern/console.c 为 kern/print.c 提供函数 cputchar。kern/print.c 中的 cprintf 函数调用函数 vcprintf，而 vcprintf 函数调用 kern/console.c 中的 putch 函数，putch 函数才会调用 cputchar 函数。
2. 这段代码主要用来打印后检测是否满屏，如果满屏，则滚动窗口（屏幕向上平移一行，最后一行全部值为空格，并将光标置于最后一行开始位置）。
3. 参数 fmt 指向的是字符串“x %d， y %x， z %d\n”，参数 ap 指向的是参数栈，具体地说，ap 指向形参 x；题目要求的列举如图 1.5 所示。

```
vcprintf: fmt = 0xf0101744, ap = 0xf0101744 + (17 + 3) / 4 * 4
cons_putc('x')
cons_putc(' ')
va_arg: before the call, ap points to x; after the call, ap points to y
cons_putc('1')
cons_putc(',')
cons_putc(' ')
cons_putc('y')
cons_putc(' ')
va_arg: before the call, ap points to y; after the call, ap points to z
cons_putc('3')
cons_putc(',')
cons_putc(' ')
cons_putc('z')
cons_putc(' ')
va_arg: before the call, ap points to z
cons_putc('4')
cons_putc('\n')
```

图 1.5 Exercise 9 的 Question 3 答案

4. “Hello”实际上是由“H”与数字 e110 组成的，这就是因为十进制数 57616 用 16 进制数来表示便是 e110。而无符号整形数在这里则表示了一个字符串“rld”，

由于无符号整形数是占 4 个字节，而低位数字是存在低地址处。若将这四个字节看做一个字符串，则每个字节代表的就是一个字符的 ASCII 码，所以低位的 0x72 代表的是字符‘r’，而最高位的 0x00 代表就是空字符，即标识字符串的结束。于是字符串与“Wo”组成了“World”， 所以最终在屏幕上输出了“He110 World”；如果是大端机，i 应该设置为 0x726c6400。

5. 显示出来 y=1656，是个随机的数字，这是因为可变参数只有一个，而可变参数指针指向的是这一个参数存放的位置，当函数试图去在内存中寻找不存在的第二个参数的时候便会在内存中存放第一个参数之后的位置中去取，而这个位置存放的内容我们无法确定，因此打印出来的便是一个随机的数字。
6. 利用 va_arg、va_start 等函数自定义取参数函数，用一个队列存储形参，这样可以反向取形参，也就是保证位置靠前的形参先被取出。

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

实际上，一个字符的打印所调用的函数依次为：cprintf → vcprintf → vprintfmt → putch → cputchar → cons putc→ cga_putc。打印信息时调用的是 cprintf，如果是彩色输出，在调用 cprintf 时就应该确定了信息打印时的颜色，但如何传递下去呢？借鉴了网上相关资料后，选择了一个牺牲封闭性但实现及其简单的方法，在 vcprintf 中如果遇到了设置颜色的参数“%C”后，便设置全局变量“ch_color”为对应的参数，在调用 cga_putc 函数时，只需读取全局变量 ch_color 的值即可获取当前需要输出的颜色信息。添加代码如图 1.6 和图 1.7 所示。（该 Challenge 在 Lab2 最终实现）

```
232         // Color
233         case 'C':
234             ch_color = getuint(&ap, lflag);
235             break;
236
```

图 1.6 在 lib/printfmt.c 中添加彩色输出相关代码

```
148 void
149 cga_putc(int c)
150 {
151     c = c + (ch_color << 8);
152     // if no attribute given, then use black on white
153     if (!(c & ~0xFF))
154         c |= 0x0700;
155         //c |= 0x7100;
156
```

图 1.7 在 kern/console.c 中添加彩色输出相关代码

在调用时，只需利用 cprintf 的“%C”参数，即可设置输出的字体颜色。在图 1.7 中可以看到，cga_putc 传入的为 int 类型的形参 c，c 的低 8 位是为了指定需要打印的字符的 ASCII 码，而高 8 位则是指定打印的颜色：c 的 15 到 12 位用于指定字符的背景颜色，颜色由 3 位 RGB 颜色代码+1 位是否高亮指定，11 到 8 位用于指定字符的前景颜色。为了方便使用，在 inc/stdio.h 中添加一些设置好的全局变量如图 1.8。

```
10 #define COLOR_WHT 7
11 #define COLOR_BLK 1
12 #define COLOR_GRN 2
13 #define COLOR_RED 4
14 #define COLOR_GRY 8
15 #define COLOR_YLW 15
16 #define COLOR_ORG 6
17 #define COLOR_PUR 12
18 #define COLOR_CYN 11
```

图 1.8 颜色设置全局变量

在 kern/monitor.c 中修改 monitor 函数中的 cprintf 添加“%C”参数，使得 monitor 能够彩色输出，如图 1.9 所示。

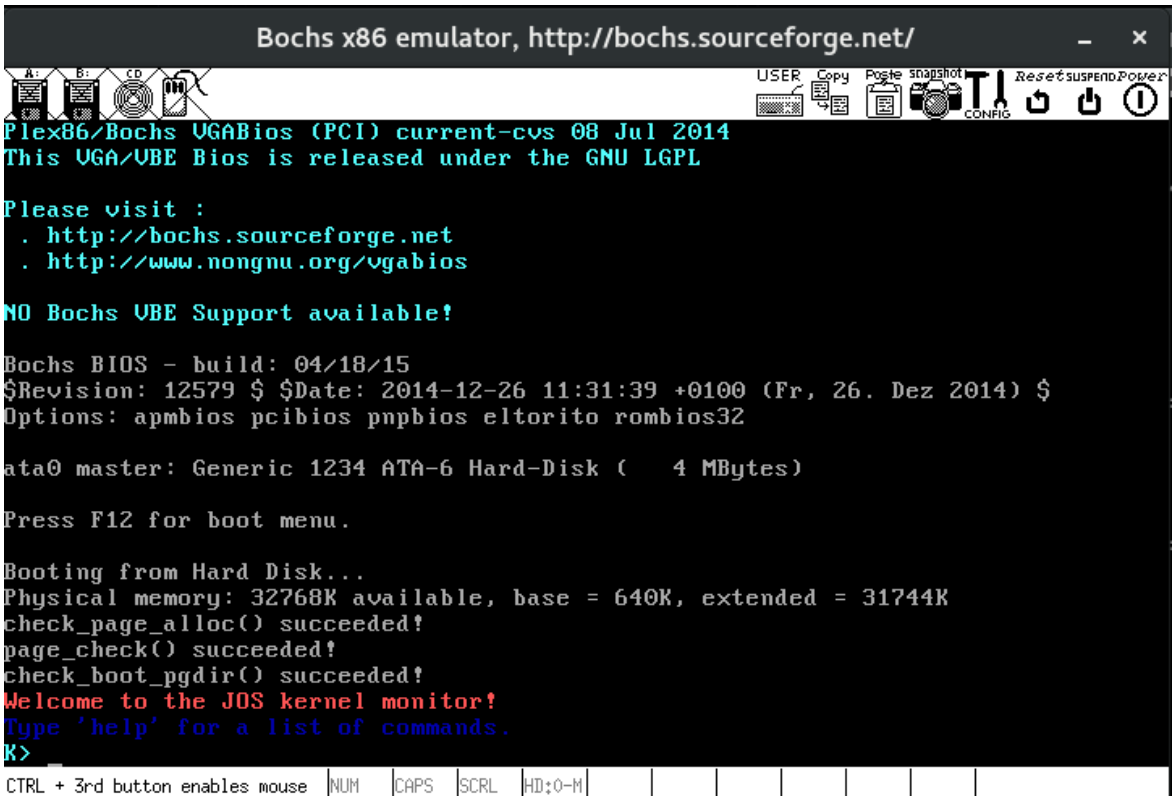

图 1.9 彩色输出效果

1.3.3 The Stack

Exercise 10. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

内核初始化栈的代码在 kern/entry.S 中，如图 1.10 和图 1.11 所示。

```
56 relocated:
57
58     # Clear the frame pointer register (EBP)
59     # so that once we get into debugging C code,
60     # stack backtraces will be terminated properly.
61     movl    $0x0,%ebp           # nuke frame pointer
62
63     # Set the stack pointer
64     movl    $(bootstacktop),%esp
65
66     # now to C code
67     call    i386_init
```

图 1.10 内核初始化栈的代码

```
83 ###################################################################
84 # boot stack
85 ###################################################################
86     .p2align    PGSHIFT     # force page alignment
87     .globl      bootstack
88 bootstack:
89     .space      KSTKSIZE
90     .globl      bootstacktop
91 bootstacktop:
92
```

图 1.11 bootstack 的定义

通过图 1.11 可以知道，bootstack 大小为 KSTKSIZE（定义在 inc/memlayout.h），bootstacktop 指向的是这片区域的该地址端，由于 JOS 中栈是从高地址向低地址生长，所以该位置就是栈底，初始化时赋给了%esp。

Exercise 11. 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 in Bochs, and examine what happens each time it gets called after the kernel starts. There are two ways you can set this breakpoint: with the `b` command and a physical address, or with the `vb` command, a segment selector (use 8 for the code segment), and a virtual address. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

学习汇编和计算机系统基础时对 C 的函数调用过程都进行了深入的研究，所以这个 Exercise 也是直接就过了。

Exercise 12. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `gmake grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

根据函数调用时的压栈可知，当进入函数体之后，当前函数的%ebp 设置成了调用了本函数的过程所在的栈指针，这时可以通过 0x0[%ebp]可以读出上个函数的栈指针，通过 0x4[%ebp]可以读出返回地址%eip，通过 0x8[%ebp]可以读出第一个参数，以此类推读出所有参数。根据此思路，可以完成函数 mon_backtrace 如图 1.12。

```
59 int
60 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
61 {
62     uint32_t ebp;
63     ebp = read_ebp();
64     cprintf("Stack backtrace:\n");
65     while(ebp)
66     {
67         cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp, *((uint32_t *)ebp + 1),
68 ____|___|___*((uint32_t *)ebp + 2), *((uint32_t *)ebp + 3), *((uint32_t *)ebp + 4), *((uint32_t *)ebp + 5), *((uint32_t *)ebp + 6));
69         ebp = *(uint32_t *)ebp;
70     }
71     return 0;
72 }
```

图 1.12 函数 mon_backtrace 代码

在 monitor 中添加相关的指令信息后，启动 Bochs，JOS 的运行截图如图 1.13 所示。

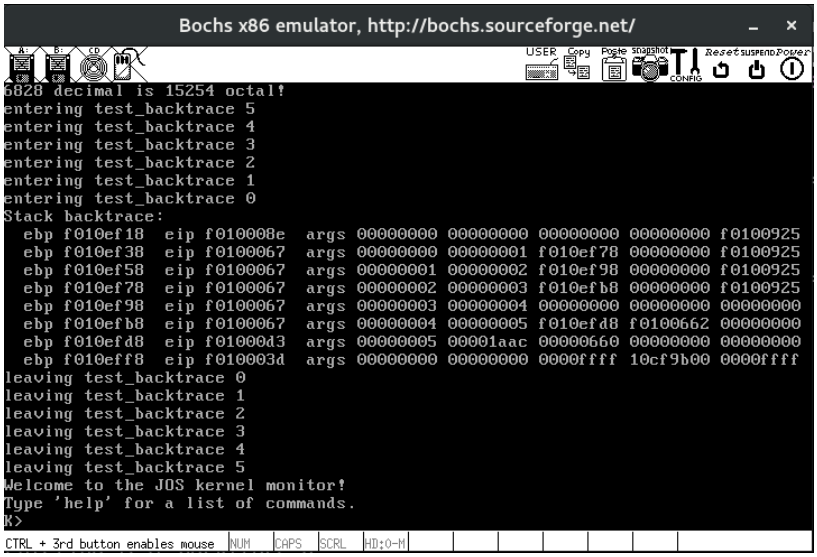

图 1.13 JOS 在 Bochs 中的运行截图

至此，Lab 1 的所有 Exercise 都已完成，其中只有两个 Exercise 要求写代码实现，而且还较为简单。完成后， 执行“make grade”进行评分，结果如图 1.14 所示。显然，评分为满分，表示 Lab 1 已经通过。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab1]
└──╼ $ make grade
make all
make[1]: Entering directory '/home/dracula/Documents/IT_Study/MIT-JOS/code/lab1'
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 508 bytes (max 510)
+ mk obj/kern/bochs.img
make[1]: Leaving directory '/home/dracula/Documents/IT_Study/MIT-JOS/code/lab1'
sh ./grade.sh
gmake[1]: Entering directory '/home/dracula/Documents/IT_Study/MIT-JOS/code/lab1'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/dracula/Documents/IT_Study/MIT-JOS/code/lab1'
Printf: OK (s)
Backtrace: Count OK, Args OK (s)
Score: 50/50
```

图 1.14 Lab 1 评分测试结果 zx

2 Memory Management

Exercise 1. Modify your stack backtrace function to display, for each EIP, the function name, source file name, and line number corresponding to that EIP. To help you we have provided `debuginfo_eip`, which looks up `eip` in the symbol table and is defined in `kern/kdebug.c`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at init.s.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
Stack backtrace:
kern/monitor.c:74: mon_backtrace+10
  ebp f0119ef8  eip f01008ce  args 00000001 f0119f20 00000000 00000000 2000000a
kern/monitor.c:143: monitor+10a
  ebp f0119f78  eip f01000e5  args 00000000 f0119fac 00000275 f01033cc ffffffc
kern/init.c:78: _panic+51
  ebp f0119f98  eip f010133e  args f01033ab 00000275 f01033cc f0103473 f01030bc
kern/pmap.c:711: page_check+9e
  ebp f0119fd8  eip f0100082  args f0102d20 00001aac 000006a0 00000000 00000000
kern/init.c:36: i386_init+42
  ebp f0119ff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
```

The `read_eip()` function may help with the first line. You may find that the some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

stab 节为 ELF 文件结构中的符号表部分，该部分的功能是为程序调错时提供报错和调试信息；stabstr 则是符号表中的字符串部分，和 stab 节一起配合打印使用。JOS 解析 stab 节使用的数据结构在 inc/stab.h 中，如图 2.1 所示。

```
42 // Entries in the STABS table are formatted as follows.
43 struct Stab {
44     uint32_t n_strx;    // index into string table of name
45     uint8_t n_type;         // type of symbol
46     uint8_t n_other;        // misc info (usually empty)
47     uint16_t n_desc;        // description field
48     uintptr_t n_value;  // value of symbol
49 };
```

图 2.1 stab 数据结构

首先回答 Exercise 中的问题，stab 节的位置信息__STAB_*等宏是链接器在链接时得到的，用来初始化 stabs 和 stab_end 变量。然后，看下 stab_binsearch 函数，实际上这是一个在 stab 节指定范围内利用二分查找法寻找指定类型和地址的表项的函数，理解到这里基本上就能知道怎么使用 stab_binsearch 函数了。接着，看下需要完成的 debuginfo_eip 函数，该函数就是在 stab 中查找给定 eip 对应的所有调试信息，其主体已经给出，只需要完成查找源文件的行号和函数参数的个数即可，按照备注

给出的提示很容易就能完成。

最后，修改 kern/monitor.c 中 Lab1 完成的 mon_backtrace 函数为如图 2.2 所示。

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t ebp, eip;
    struct Eipdebuginfo info;
    char func_name[36];
    eip = read_eip();
    ebp = read_ebp();
    cprintf("%CStack backtrace:\n", COLOR_GRN);
    while(ebp) {
        debuginfo_eip(eip, &info);
        strncpy(func_name, info.eip_fn_name, info.eip_fn_namelen);
        func_name[info.eip_fn_namelen] = '\0';
        cprintf("%C%s:%d: %s+%x\n", COLOR_YLW, info.eip_file, info.eip_line, func_name, eip - info.eip_fn_addr);
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n%C", ebp, *((uint32_t *)ebp + 1),
            *((uint32_t *)ebp + 2), *((uint32_t *)ebp + 3), *((uint32_t *)ebp + 4), *((uint32_t *)ebp + 5), *((uint32_t *)ebp + 6), COLOR_CYN);
        eip = *((uint32_t *)ebp + 1);
        ebp = *(uint32_t *)ebp;
    }
    return 0;
}
```

图 2.2 修改后的 mon_backtrace 函数

这里需要注意的是 read_eip 函数，如果不做改动，该函数会通过 ebp 来获取 eip（方法就像 Lab1 中提到的那样），而不是 eip 寄存器中的内容，这样就会带来一个问题，就是当前函数（mon_backtrace）不能被 trace 到，如图 2.3 所示

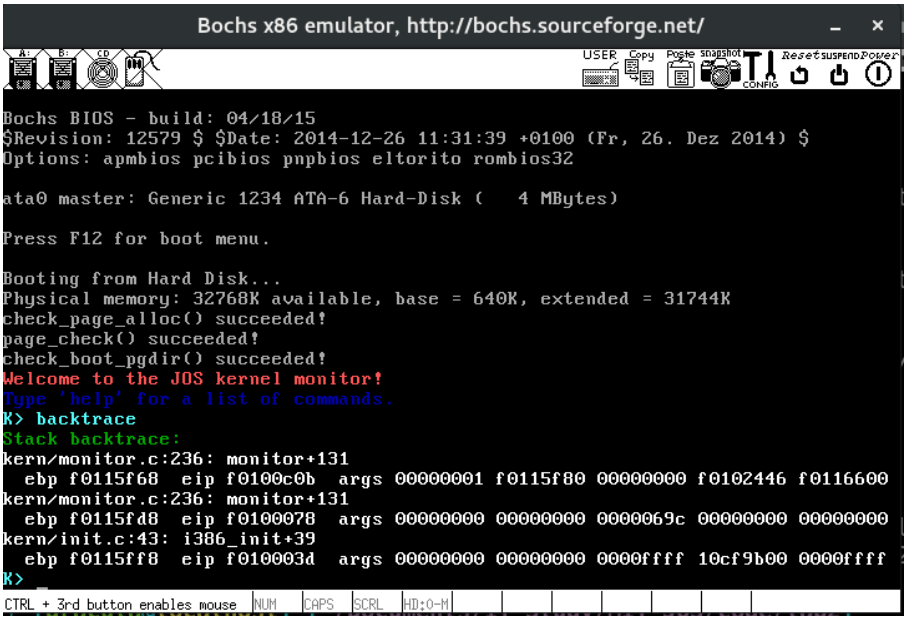

图 2.3 read_eip 修改前 backtrace 的打印信息

询问老师后，为了解决这个问题，需要将 read_eip 函数声明为非内联的函数，具体的方法就是在 kern/monitor.c 中添加如下代码：

unsigned read_eip() __attribute__((noinline));

添加声明后，再次调用的 read_eip 函数就会读取%eip 寄存器中的值作为 eip，也就是“真正的”eip。修改后，再次启动 Bochs，执行 backtrace 命令，就可以看到当前函数确实为 mon_backtrace，如图 2.4 所示。

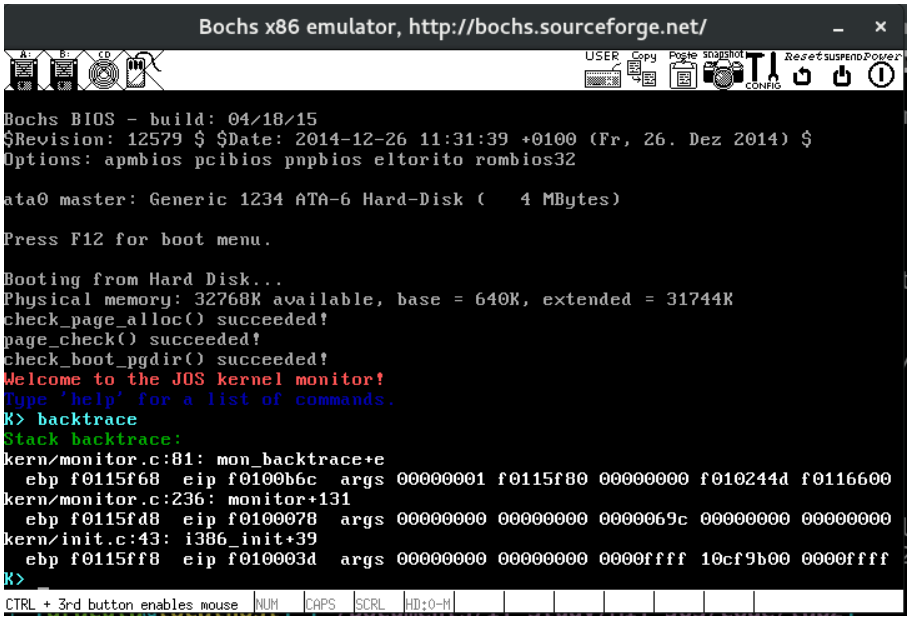

图 2.4 read_eip 修改后 backtrace 的打印信息

2.1 Physical Page Management

Exercise 2. In the file `kern/pmap.c`, you must implement code for the following functions.

```
boot_alloc()
page_init()
page_alloc()
page_free()
```

You also need to add some code to `i386_vm_init()` in `pmap.c`, as indicated by comments there. For now, just add the code needed before the call to `check_page_alloc()`.

`check_page_alloc()` tests your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your own assumptions are correct.

先来看下调用 i386_init()函数以前内存的 layout，如图 2.5（图片来自课设资料）。在 Lab1 中提到过，内核放到了 0x100000 这个位置，直到 end（链接器得到）。

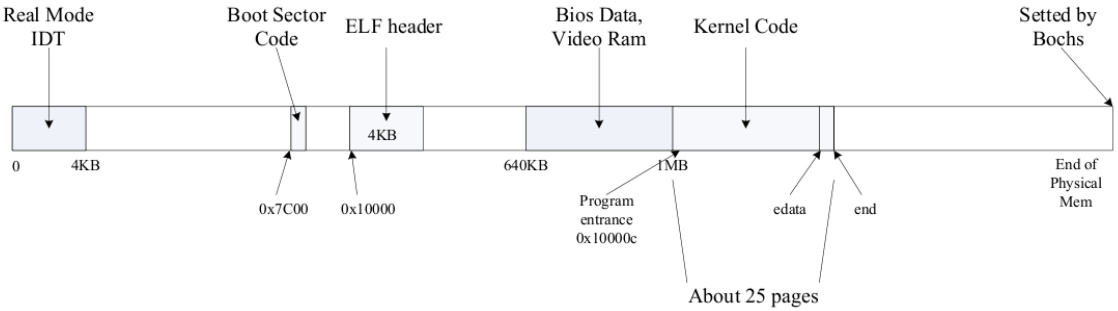

图 2.5 调用 i386_init()函数以前内存的 layout

在建立物理页面对应的 Page* 链表时，需要为这个链表分配实际的物理内存空间，在二级页地址映射机制中，系统还需要一个页目录存下所有二级页表的地址，

这个也是需要操作系统预先分配空间的，分配完成后的物理内存布局如图 2.6 所示。

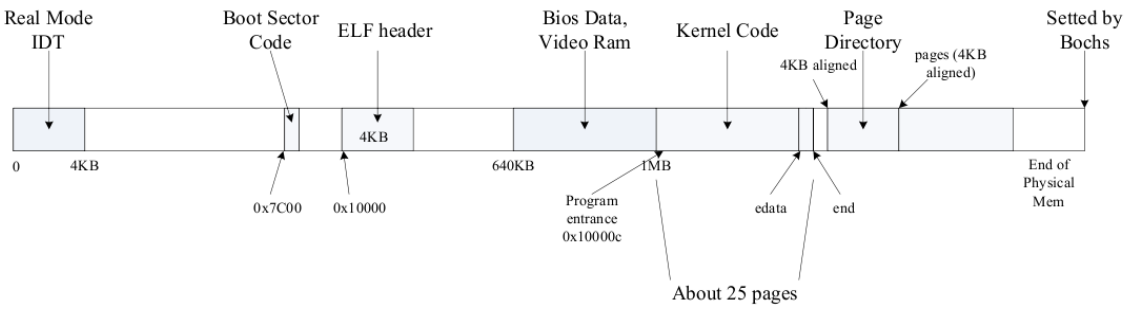

图 2.6 放入页面管理内容后的物理内存布局

然后，再看看如何完成这个 Exercise 需要完成的函数。首先，是 boot_alloc 函数，完成该函数需要知道全局变量 boot_freemem 的含义，boot_freemem 是当前可用内存的开始地址（一开始等于 end）。理解了 boot_freemem，按照注释一步一步写出代码即可完成 boot_alloc 函数，如图 2.7

```
105 static void*
106 boot_alloc(uint32_t n, uint32_t align)
107 {
108     extern char end[];
109     void *v;
110
111     // Initialize boot_freemem if this is the first time.
112     // 'end' is a magic symbol automatically generated by the linker,
113     // which points to the end of the kernel's bss segment -
114     // i.e., the first virtual address that the linker
115     // did _not_ assign to any kernel code or global variables.
116     if (boot_freemem == 0)
117         boot_freemem = end;
118
119     // LAB 2: Your code here:
120     //  Step 1: round boot_freemem up to be aligned properly
121     boot_freemem = ROUNDUP(boot_freemem, align);
122     //  Step 2: save current value of boot_freemem as allocated chunk
123     v = boot_freemem;
124     //  Step 3: increase boot_freemem to record allocation
125     boot_freemem += n;
126     //  Step 4: return allocated chunk
127     return v;
128 }
```

图 2.7 boot_alloc 函数

接下来考虑 page_init 函数，该函数的具体工作就是为每个物理页面建立对应的链表节点，然后将操作系统占用的空间以及预留空间从链表中除掉。根据该函数注释的提示，需要去掉的内存空间有：Page0—存放中断向量表 IDT 以及 BIOS 的相关载入程序，[IOPHYSMEM， EXTPHYSMEM)—输入输出所需空间，[EXTPHYSMEM， end)—操作系统内核，[PADDR(boot pgdir)， PADDR(boot pgdir) + PGSIZE)—存放页目录，[PADDR(pages)， boot freemem)—存放 pages。除了 Page0，

后面 4 个空间实际上是连续分布的，即[IOPHYSMEM， boot freemem)。除此之外，还需了解一个宏 PPN，可以将物理地址转化为其在 pages 数组中的页号。到这里，基本可以完成 page_init 函数了，如图 2.8 所示。

```
422 void
423 page_init(void)
424 {
425     // The example code here marks all pages as free.
426     // However this is not truly the case.  What memory is free?
427     //  1) Mark page 0 as in use.
428     //     This way we preserve the real-mode IDT and BIOS structures
429     //     in case we ever need them.  (Currently we don't, but...)
430     //  2) Mark the rest of base memory as free.
431     //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM).
432     //     Mark it as in use so that it can never be allocated.
433     //  4) Then extended memory [EXTPHYSMEM, ...).
434     //     Some of it is in use, some is free. Where is the kernel?
435     //     Which pages are used for page tables and other data structures?
436     //
437     // Change the code to reflect this.
438     int i;
439     int lower_ppn = PPN (IOPHYSMEM);
440     int upper_ppn = PPN (ROUNDUP(boot_freemem, PGSIZE));
441     LIST_INIT(&page_free_list);
442     for (i = 0; i < npage; i++) {
443         pages[i].pp_ref = 0;
444         if (i == 0)
445             continue;
446         if (lower_ppn <= i && i < upper_ppn)
447             continue;
448         LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
449     }
450 }
```

图 2.8 page_init 函数

page_alloc 函数、page_free 函数以及上面的 page_init 函数都需要了解 JOS 中的链表实现，具体来说就是几个宏的使用，看懂不难。看懂这些宏，page_alloc 函数和 page_free 函数就十分容易了，代码分别如图 2.9 和图 2.10 所示。

```
500 int
501 page_alloc(struct Page **pp_store)
502 {
503     // Fill this function in
504     if (LIST_FIRST(&page_free_list) != NULL) {
505         *pp_store = LIST_FIRST(&page_free_list);
506         // Remove the page from page_free_list
507         LIST_REMOVE(*pp_store, pp_link);
508         page_initpp(*pp_store);
509         return 0;
510     }
511     else {
512         return -E_NO_MEM;
513     }
514 }
```

图 2.9 page_alloc 函数

```
532 void
533 page_decref(struct Page* pp)
534 {
535     if (--pp->pp_ref == 0)
536         page_free(pp);
537 }
538
```

图 2.10 page_free 函数

2.2 Virtual Memory

Exercise 3. Read chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Although JOS relies most heavily on page translation, you will also need a basic understanding of how segmentation works in protected mode to understand what's going on in JOS.

关于分段的内容，没有看原版的 80386 的参考资料，而是阅读了华科课程设计的相关资料的“Chapter4.pdf”，然后结合上学期操作系统课程上所学习到的内容，基本可以理解 JOS 的分段机制。

2.2.1 Virtual， Linear， and Physical Addresses

Exercise 4. Review the debugger section in the Bochs user manual, and make sure you understand which debugger commands deal with which kinds of addresses. In particular, note the various `vb`, `lb`, and `pb` breakpoint commands to set breakpoints at virtual, linear, and physical addresses. The default `b` command breaks at a *physical* address. Also note that the `x` command examines data at a *linear* address, while the command `xp` takes a physical address. Sadly there is no `xv` at all.

这一部分内容，仍然看的是课设资料“Chapter4.pdf”。对于设置断点的调试指令“vb、lb 和 pb”，用得较多的是 vb 和 pb，根据所知道的地址类型灵活选择即可。

C type	Address type
`T*`	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

Question

1. Assuming that the following JOS kernel code compiles correctly and doesn't crash, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

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

这一部分主要讲了 JOS 中 Virtual Address 和 Physical Address 对应的 C 语言的类型。至于这个 Question，因为内核中的代码使用的都是内核虚拟地址，所以显然 x 的类型应该为“uintptr_t”。

2.2.2 Page Table Management

Exercise 4. In the file `kern/pmap.c`, you must implement code for the following functions.

```
pgdir_walk()
boot_map_segment()
page_lookup()
page_remove()
page_insert()
```

`page_check()`, called from `i386_vm_init()`, tests your page table management routines. You should make sure it reports success before proceeding.

实际上这个 Exercise 应该是“Exerci 5”，“Exercise 4”前面已经有过了。

可以说，这个 Exercise 的内容基本代表了 Lab 2 的内容，共需要完成 5 个关于页表管理的函数。在正式实现这 5 个函数之前，最好阅读一下在 kern/pmap.h 中定义好的一组宏，用于页表地址与物理地址、物理页号之间的转换。

首先，考虑 pgdir_walk 函数，该函数的主要目的是实现虚地址到物理地址的转换。根据给定的虚地址，如果有对应的物理页面，则返回二级页表中的页表项；如果没有对应的物理页面，则根据参数 create 决定是否分配一个物理页面给该虚地址。按照注释的提示写代码即可，需要注意的 memset 用的是虚拟地址，所以需要用宏 page2kva 转化为内核虚拟地址；但是在页目录和页表项中的需要是物理地址，因为是 CPU 直接访问这些地址。最终，pgdir_walk 函数的代码如图 2.11 所示。

```
552 pte_t *
553 pgdir_walk(pde_t *pgdir, const void *va, int create)
554 {
555     // Fill this function in
556     // Notice: pte_t is physical address
557     struct Page * page;
558     if ((pgdir[PDX(va)] & PTE_P) == 1) {
559         // If the page exists(P = 1)
560         return (pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])) + PTX(va);
561     }
562     else {
563         if (create == 0)
564             return NULL;
565         else {
566             // Allocate a page for page table
567             if (page_alloc(&page) == 0) {
568                 // Initial the page for page table
569                 memset(page2kva(page), 0, PGSIZE);
570                 page->pp_ref = 1;
571                 // Modify contents in page directory
572                 pgdir[PDX(va)] = page2pa(page) | PTE_U | PTE_W | PTE_P ;
573                 // Wrong permission will cause unexcepted result
574                 //pgdir[PDX(va)] = page2pa(page) | PTE_P ;
575                 return (pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])) + PTX(va);
576             }
577             else
578                 return NULL;
579         }
580     }
581 }
```

图 2.11 pgdir_walk 函数

然后，考虑 boot_map_segment 函数，该函数用于将线性地址 la 开始的 size 大小的空间映射到物理页面 pa 开始的同样大小的空间。显然，该函数的关键是利用 pgdir_walk 函数得到对应的二级目录页表项的地址，函数的代码如图 2.12 所示。

```
630 static void
631 boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, physaddr_t pa, int perm)
632 {
633     // fill this function in
634     uint32_t i;
635     pte_t *pte;
636     for (i = 0; i < size; i += PGSIZE) {
637         pte = pgdir_walk(pgdir, (void *)(la + i), 1);
638         *pte= (pa + i) | perm | PTE_P;
639     }
640 }
```

图 2.12 boot_map_segment 函数

page_lookup 函数用查找指定虚地址对应的 page 数组元素的地址，仍然是利用 pgdir_walk 函数来查找，然后利用宏 pa2page 将其转化为 Page 数组地址，如图 2.13。

```
652 struct Page *
653 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
654 {
655     // Fill this function in
656     pte_t *pte;
657     pte = pgdir_walk(pgdir, va, 0);
658     if (pte == NULL)
659         return 0;
660     if (pte_store != NULL)
661         *pte_store = pte;
662     return pa2page(*pte);
663
664 }
```

图 2.13 page_lookup 函数

page_remove 函数则是负责在页目录中删除指定虚地址对应的二级页表项目录，利用刚才完成的 page_lookup 函数和 JOS 已经实现的 page_decref 函数，最后按照注释要调用 tlb_invalidate 函数刷新页表，函数代码如图 2.14 所示。

```
681 void
682 page_remove(pde_t *pgdir, void *va)
683 {
684     // Fill this function in
685     struct Page* page;
686     pte_t *pte;
687     page = page_lookup(pgdir, va, &pte);
688     if (page != NULL) {
689         page_decref(page);
690         *pte = 0;
691     }
692     tlb_invalidate(pgdir, va);
693     return;
694
695 }
```

图 2.14 page_remove 函数

page_insert 函数实际上是将物理页 pp 映射到虚拟地址 va，代码如图 2.15 所示。

```
602 int
603 page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
604 {
605     // Fill this function in
606     pte_t *pte;
607     pte = pgdir_walk(pgdir, va, 1);
608     if (pte != NULL) {
609         pp->pp_ref += 1;
610         if ((*pte & PTE_P) == 1)
611             page_remove(pgdir, va);
612         *pte = page2pa(pp) | perm | PTE_P;
613         return 0;
614     }
615     else
616         return -E_NO_MEM;
617 }
```

图 2.15 page_insert 函数

2.3 Kernel Address Space

这一部分主要是讲内核地址空间，对应的定义在 inc/memlayout.h 中。

2.3.1 Permissions and Fault Isolation

仔细阅读 inc/memlayout.h 中，关于[UTOP， ULIM)空间的详细介绍，学习和理解各段的含义和用途。

2.3.2 Initializing the Kernel Address Space

Exercise 6. Fill in the missing code in `i386_vm_init()` after the call to `page_check()`.

Your code should now pass the `check_boot_pgdir()` check.

这部分需要修改下 i386_vm_init 函数。修改前，先仔细学习了该函数的前半部分，包括创建并初始化页目录、pages 数组等。需要添加的部分主要是将虚地址 UPAGES、KSTACKTOP、KERNBASE 等映射到相应的物理地址，代码如图 2.16。

```
201     boot_map_segment(pgdir, UPAGES, ROUNDUP(npage * sizeof(struct Page), PGSIZE), PADDR(pages), PTE_U | PTE_P);
202     boot_map_segment(pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W | PTE_P);
203     boot_map_segment(pgdir, KERNBASE, 0xffffffff - KERNBASE + 1, 0, PTE_W | PTE_P);
```

图 2.16 i386_vm_init 函数

Answer these questions:

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

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. After `check_boot_pgdir()`, `i386_vm_init()` maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?
3. (From Lecture 4) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? Why?
5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

1. 根据 i386_vm_init 函数中的三次调用 boot_map_segment 函数，这题不难解决，篇幅原因不列出具体的表格了。
2. This mapping is necessary. In function i386_vm_init()，after code "lcr0(cr0)" paging is turned on while all entry's segment address in gdt are 0x10000000(-KERNBASE). So at this point， mapping is "KERNBASE+x => x => unknown"(Format: va => la => pa). Before paging is turned on， the la is equal to pa. To keep what kernel access before and after paging is the same， make that "pgdir[0] = pgdir[pdx(kernbase)]" so

that mapping is "KERNBASE+x => x => x"(before paging the map is "KERNBASE+x => x => x"). Next asm code will reload gdt whose entry's segment address is zero， so the final mapping is "KERNBASE+x => KERNBASE+x => x". Because kernel is mapped using function boot_map_segment()， so pgdir[0] doesn't matter any more(pgdir[0] = 0). I guess in asm code， it just use first four MB. （写课设的时候用英文写的，太长了，不翻译回来了）。

3. 肯定不能，否则会导致不可预料的问题。
4. 2^32B = 4GB
5. 一个页目录和 npage 个页表，总共为 PGSIZE + npage * PGSIZE = 4KB + 1024 * 4KB = 1025 * 4KB，大约 4MB。

Challenge! Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter `'showmappings 0x3000 0x5000'` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

这一系列 Challenge 只完成了第一个，即扩展了“showmappings”指令。

扩展指令主要是完成对应的 mon_showmappins 函数，该函数逻辑上很简单，只是需要判断的地方比较，通过 pgdir_walk 函数可以得到对应的映射关系，mon_showmappins 函数的代码如图 2.17 所示。

```
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc == 3) {
        uint32_t lva = ROUNDDOWN(strtol(argv[1], 0, 0), PGSIZE);
        uint32_t hva = ROUNDUP(strtol(argv[2], 0, 0), PGSIZE);
        if (hva >= lva) {
            uint32_t i;
            pte_t *pte;
            for (i = lva; i <= hva; i += PGSIZE)
            {
                pte = pgdir_walk(boot_pgdir, (void *)i, 0);
                cprintf("%C0x%x %C- %C0x%x    ", COLOR_GRN, i, COLOR_CYN, COLOR_GRN, i + PGSIZE);
                if (pte != NULL && (*pte & PTE_P)) {
                    cprintf("%Cmapped %C0x%x  ", COLOR_YLW, COLOR_PUR, PTE_ADDR(*pte));
                    if (*pte & PTE_U)
                        cprintf ("%Cuser: ", COLOR_BLK);
                    else
                        cprintf ("%Ckernel: ", COLOR_BLK);
                    if (*pte & PTE_W)
                        cprintf ("%Cread/write", COLOR_PUR);
                    else
                        cprintf ("%Cread only", COLOR_PUR);
                    cprintf("\n%C", COLOR_CYN);
                }
                else {
                    cprintf("%Cnot mapped\n%C", COLOR_RED, COLOR_CYN);
                }
            }
        }
        else
            cprintf("%Cshowmappings: Invalid address\n%C", COLOR_RED, COLOR_CYN);
    }
    else
        cprintf("%CUsage: showmappings LOWER_VIRTUAL_ADDR HIGHER_VIRTUAL_ADDR\n%C", COLOR_GRN, COLOR_CYN);
    return 0;
}
```

图 2.17 mon_showmappins 函数

启动 Bochs，执行 showmappings 指令，运行截图如图 2.18 所示。

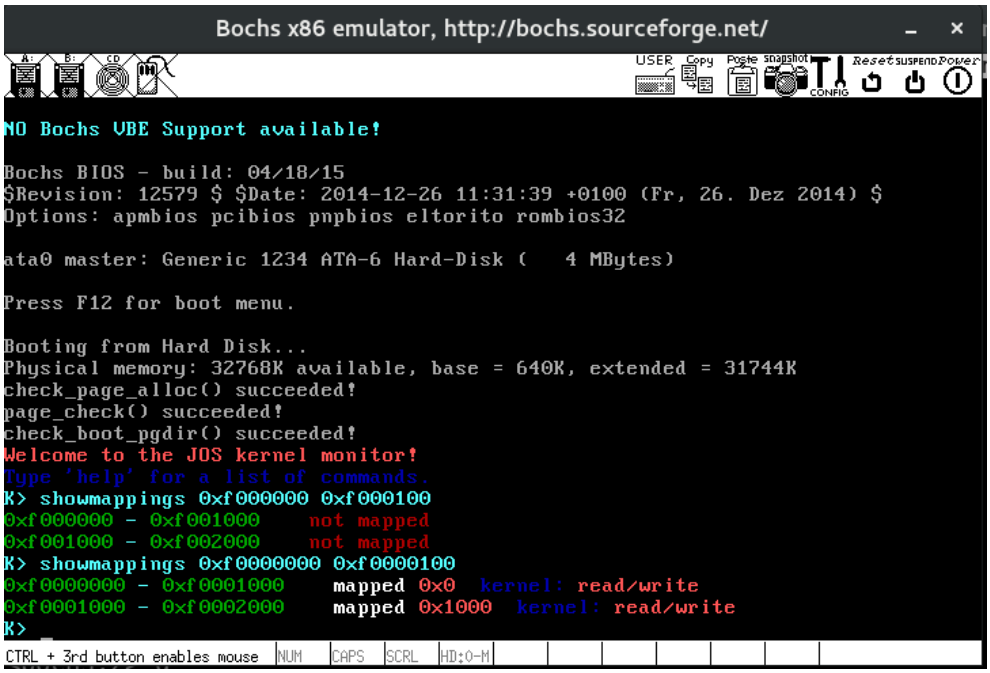

图 2.18 showmappings 指令运行截图

2.3.3 Address Space Layout Alternatives

Challenge! Extend the JOS kernel monitor with commands to allocate and free pages explicitly, and display whether or not any given page of physical memory is currently allocated. For example:

```
K> alloc_page
     0x13000
K> page_status 0x13000
     allocated
K> free_page 0x13000
K> page_status 0x13000
     free
```

Think of other commands or extensions to these commands that may be useful for debugging, and add them.

这个 Challenge 是整个实验遇到最简单的 Challenge，需要做的就是简单地将前面完成的页表管理函数封装一下即可。启动 Bochs，运行截图如图 2.19 所示。

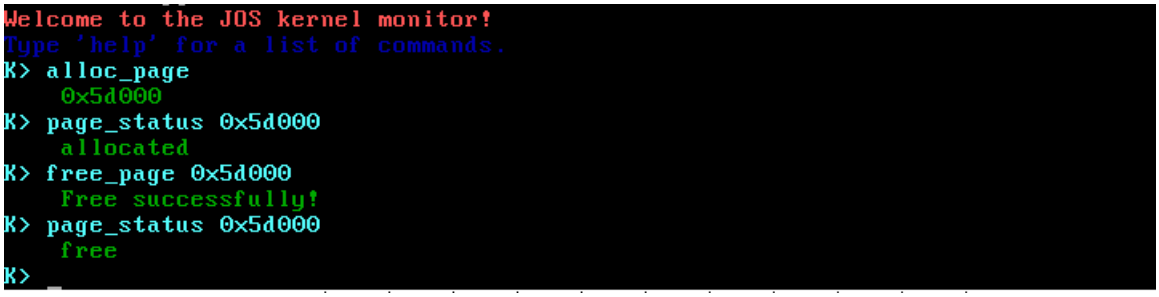

图 2.19 alloc、status、free_page 运行截图

对应的 mon_*函数代码如图 2.20 所示。

```
135 int
136 mon_alloc_page(int argc, char **argv, struct Trapframe *tf)
137 {
138     struct Page *page;
139     if (page_alloc(&page) == 0) {
140         page->pp_ref += 1;
141         cprintf("    %C0x%x\n%C", COLOR_GRN, page2pa(page), COLOR_CYN);
142     }
143     else
144         cprintf("    %CAllocate failed!\n%C", COLOR_RED, COLOR_CYN);
145     return 0;
146 }
147
148 int
149 mon_free_page(int argc, char **argv, struct Trapframe *tf)
150 {
151     if (argc == 2) {
152         struct Page *page = pa2page(strtol(argv[1], 0, 0));
153         if (page->pp_ref == 1) {
154             page_decref(page);
155             cprintf("    %CFree successfully!\n%C", COLOR_GRN, COLOR_CYN);
156         }
157         else
158             cprintf("    %CFree failed!\n%C", COLOR_RED, COLOR_CYN);
159     }
160     else
161         cprintf("%CUsage: free_page PHYSIC_ADDR\n%C", COLOR_GRN, COLOR_CYN);
162     return 0;
163 }
164
165 int
166 mon_page_status(int argc, char **argv, struct Trapframe *tf)
167 {
168     if (argc == 2) {
169         struct Page *page = pa2page(strtol(argv[1], 0, 0));
170         if (page->pp_ref > 0)
171             cprintf("    %Callocated\n%C", COLOR_GRN, COLOR_CYN);
172         else
173             cprintf("    %Cfree\n%C", COLOR_GRN, COLOR_CYN);
174     }
175     else
176         cprintf("%CUsage: page_status PHYSIC_ADDR\n%C", COLOR_GRN, COLOR_CYN);
177     return 0;
178 }
```

图 2.20 mon_*函数

至此，Lab 2 的所有 Exercise 都已完成，并完成了两个 Challenge。完成后， 执行“sh grade.sh”进行评分，结果如图 2.21 所示。显然，评分为满分，表示 Lab 2 已经通过。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab2]
└──╼ $ sh grade.sh
gmake: Nothing to be done for 'all'.
Page directory: OK (s)
Page management: OK (s)
Score: 50/50
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab2]
└──╼ $
```

图 2.21 Lab 2 评分测试结果

但是在实际做实验的时候，一开始，Lab 2 的评分虽然都已经通过，但是在 Bochs 启动 JOS 时，却一直不能正常启动，总是跳回第一条指令执行，如图 2.22 所示。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab2]
└──╼ $ bochs -q
========================================================================
                       Bochs x86 Emulator 2.4.6
             Build from CVS snapshot, on February 22, 2011
                   Compiled at Feb  6 2017, 00:14:56
========================================================================
00000000000i[     ] reading configuration from .bochsrc
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> c
Physical memory: 32768K available, base = 640K, extended = 31744K
check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
(0).[1567650083] [0x000000000010247a] 0008:f010247a (unk. ctxt): mov gs, ax                ; 8ee8
Next at t=1567650084
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:2> c
Physical memory: 32768K available, base = 640K, extended = 31744K
check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
(0).[3147264207] [0x000000000010247a] 0008:f010247a (unk. ctxt): mov gs, ax                ; 8ee8
Next at t=3147264208
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:3>
```

图 2.22 Lab 2 的 Bug 截图

寻找了大约一上午的 Bug，发现是在 pgdir_walk 这个函数中对页目录项设置权限时只设置的“PTE_P”，导致内核对申请到的物理页没有写权限，从而导致了 JOS 崩溃。错误代码为 inc/pmap.c 的第 581 行：“pgdir[PDX(va)] = page2pa(page) | PTE_P ;”将其修改为“pgdir[PDX(va)] = page2pa(page) | PTE_U | PTE_W | PTE_P ;”，Bug 便解决了，Bochs 可以正常启动 JOS 了，如图 2.23 所示。

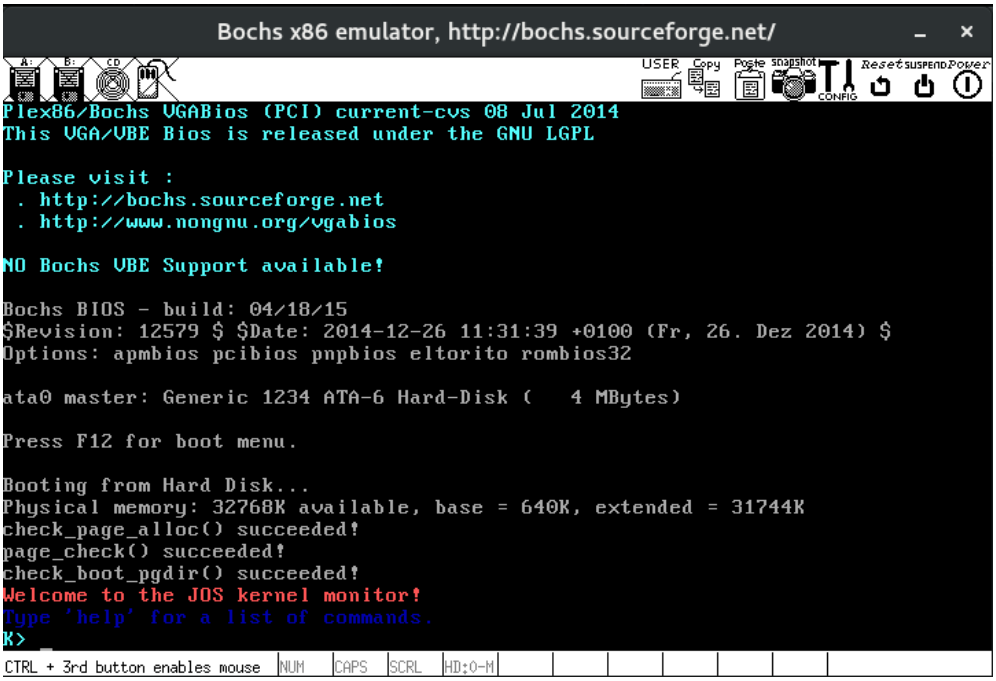

图 2.23 Bochs 正常启动 JOS

3 User Environments

3.1 User Environments and Exception Handling

3.1.1 Environment State

这一部分主要就是对 inc/env.h 中定义的结构体 Env 进行描述，MIT 的课程网页上给出的说明真的十分详细，这里不重复了。

3.1.2 Allocating the Environments Array

Exercise 1. Modify `i386_vm_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure, laid out consecutively in the kernel's virtual address space starting at address `UENVS` (defined in `inc/memlayout.h`). You should allocate and map this array the same way as you did the `pages` array.

这个 Exercise 因为有了 Lab 2 设置 pages 的经历，所以做起来十分简单，只需照着申请 pages 和绑定的过程，对数组 envs 进行类似的操作即可。在 kern/pmap.c 的 i386_vm_init 函数中添加如下两行代码：

envs = boot_alloc(NENV * sizeof(struct Env)， PGSIZE);

boot_map_segment(pgdir， UENVS， ROUNDUP(NENV * sizeof(struct Env)，PGSIZE)， PADDR(envs)， PTE_U | PTE_P);

这个 Exercise 就基本通过了。

3.1.3 Creating and Running Environments

Exercise 2. In the file `env.c`, finish coding the following functions:

`env_init()`:
initialize all of the `Env` structures in the `envs` array and add them to the `env_free_list`.

`env_setup_vm()`:
allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

`segment_alloc()`:
allocates and maps physical memory for an environment

`load_icode()`:
you will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.

`env_create()`:
allocate an environment with `env_alloc` and call `load_icode` load an ELF binary into it.

`env_run()`:
start a given environment running in user mode.

As you write these functions, you might find the new cprintf verb `%e` useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

这个 Exercise 的要求是完成创建和运行 environment 的一些函数，共 6 个。

首先，先来考虑 env_init 函数，跟 pages 的初始化差不多甚至更简单，不用考虑被系统占用的情况，函数代码如图 3.1 所示。

```
71 void
72 env_init(void)
73 {
74     // LAB 3: Your code here.
75     int i;
76     LIST_INIT(&env_free_list);
77     for (i = NENV - 1; i >= 0; i--) {
78         envs[i].env_id = 0;
79         envs[i].env_status = ENV_FREE;
80         LIST_INSERT_HEAD(&env_free_list, envs + i, env_link);
81     }
82 }
```

图 3.1 env_init 函数

然后考虑 env_setup_vm 函数，该函数用于为 environment 申请一个 page 作为页目录，并进行初始化。由于在 UTOP 之上的所有映射对于任何一个地址空间都是一样的，所以直接拷贝内核的页目录到对应 page，函数代码如图 3.2 所示。

```
94 static int
95 env_setup_vm(struct Env *e)
96 {
97     int i, r;
98     struct Page *p = NULL;
99
100    // Allocate a page for the page directory
101    if ((r = page_alloc(&p)) < 0)
102        return r;
103
104    // LAB 3: Your code here.
105
106    e->env_pgdir = page2kva(p);
107    e->env_cr3 = page2pa(p);
108    memmove(e->env_pgdir, boot_pgdir, PGSIZE);
109    memset(e->env_pgdir, 0, PDX(UTOP) * sizeof(pde_t));
110    p->pp_ref += 1;
111
112    // VPT and UVPT map the env's own page table, with
113    // different permissions.
114    e->env_pgdir[PDX(VPT)]  = e->env_cr3 | PTE_P | PTE_W;
115    e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_P | PTE_U;
116
117    return 0;
118 }
```

图 3.2 env_setup_vm 函数

接着考虑 segment_alloc 函数，主要用于为 environment 分配和绑定物理内存空间，需要注意的是该函数是要将物理页分配给 environment，所以需要 page_insert 函数，按照注释该函数不难理解和实现，如图 3.3 所示。

继续考虑 load_icode 函数，这个函数特别像 boot loader，需要加载 ELF 文件，所以仿照 boot/main.c 中的 bootmain 函数，ELF 文件的加载这部分还是比较容易完成的，不过需要注意的是载入 ELF 文件时要切换到用户空间，载入后切换回内核空间。此外，加载到内存空间的是可加载段，包括 BSS 和 DATA，对于 BSS 所占的物理空间要初始化为 0。最后，要设置 env 的入口地址即 env 开始运行时寄存器%eip 的值，还要调用 segment_alloc 函数为用户栈分配和绑定空间。具体的代码如图 3.4 所示。

```
201 static void
202 segment_alloc(struct Env *e, void *va, size_t len)
203 {
204     // LAB 3: Your code here.
205     // (But only if you need it for load_icode.)
206     //
207     // Hint: It is easier to use segment_alloc if the caller can pass
208     //   'va' and 'len' values that are not page-aligned.
209     //   You should round va down, and round len up.
210     int i, err;
211     struct Page *page;
212     va = ROUNDDOWN(va, PGSIZE);
213     len = ROUNDUP(len, PGSIZE);
214     for (i = 0; i < len; i += PGSIZE) {
215         err = page_alloc(&page);
216         if (err)
217             panic("segment_alloc: Allocate physical page failed. \n", err);
218         err = page_insert(e->env_pgdir, page, va + i, PTE_U | PTE_W);
219         if (err)
220             panic("segment_alloc: Map physical page failed. %e\n", err);
221     }
222 }
```

图 3.3 segment_alloc 函数

```
246 static void
247 load_icode(struct Env *e, uint8_t *binary, size_t size)
248 {
249     // LAB 3: Your code here.
250     struct Elf *ELFHDR = (struct Elf *)binary;
251     struct Proghdr *ph, *eph;
252
253     // is this a valid ELF
254     if (ELFHDR->e_magic != ELF_MAGIC)
255         panic("load_icode: Give binary is not a valid ELF.\n");
256
257     // load each program segment (ignores ph flags)
258     ph = (struct Proghdr *)((uint8_t *) ELFHDR + ELFHDR->e_phoff);
259     eph = ph + ELFHDR->e_phnum;
260     // switch to user address space
261     lcr3(e->env_cr3);
262     for (; ph < eph; ph++) {
263         if (ph->p_type == ELF_PROG_LOAD) {
264             segment_alloc(e, (void *)ph->p_va, ph->p_memsz);
265             memset((void *)ph->p_va, 0, ph->p_memsz);
266             memmove((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
267         }
268     }
269     // switch to kern address space
270     lcr3(boot_cr3);
271     e->env_tf.tf_eip = ELFHDR->e_entry;
272     // Now map one page for the program's initial stack
273     // at virtual address USTACKTOP - PGSIZE.
274
275     // LAB 3: Your code here.
276     segment_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
277 }
```

图 3.4 load_icode 函数

env_create 和 env_run 两个函数虽然实现比较简单，但是这是因为 JOS 把最关键的部分都实现好了，分别是 env_alloc 函数和 env_pop_tf 函数。这两个函数以及 Trapframe 结构体一定要看懂和读懂，对做后面的实验帮助极大，后面用到时再做详细介绍。env_create 和 env_run 两个函数的代码分别如图 3.5 和图 3.6 所示。

3.1.4 Handling Interrupts and Exceptions

Exercise 3. Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

```
289 void
290 env_create(uint8_t *binary, size_t size)
291 {
292     // LAB 3: Your code here.
293     int error;
294     struct Env *env;
295     error = env_alloc(&env, 0);
296     if (error < 0)
297         panic("env_create: Allocate environment failed. %e", error);
298     load_icode(env, binary, size);
299 }
```

图 3.5 env_create 函数

```
394 env_run(struct Env *e)
395 {
396     // LAB 3: Your code here.
397
398     if (curenv != e) {
399         curenv = e;
400         curenv->env_runs += 1;
401         lcr3(curenv->env_cr3);
402     }
403     env_pop_tf(&curenv->env_tf);
404 }
405
```

图 3.6 env_run 函数

关于中断和异常，操作系统和组原课上讲的分别从软件和硬件的角度考虑，还算比较了解，所以这个 Exercise 也是直接过了。

3.1.5 Basics of Protected Control Transfer

3.1.6 Types of Exceptions and Interrupts

3.1.7 Nested Exceptions and Interrupts

上面几个小节都是材料阅读，MIT 的课程网页上讲的十分清楚和详细，不重复。

3.1.8 Setting Up the IDT

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S` should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapentry.S` (using those macros) for each trap defined in `inc/trap.h`. You will also need to modify `idt_init()` to initialize the `idt` to point to each of these entry points defined in `trapentry.S`; the `SETGATE` macro will be helpful here.

Hint: your code should perform the following steps:

1. push values to make the stack look like a struct Trapframe
2. load GD_KD into `%ds` and `%es`
3. `pushl %esp` to pass a pointer to the Trapframe as an argument to trap()
4. `call trap`
5. pop the values pushed in steps 1-3
6. `iret`

Consider using the `pushal` and `popal` instructions; they fit nicely with the layout of the `struct Trapframe`.

Test your trap handling code using some of the test programs in the `user` directory that cause exceptions before making any system calls, such as `user/divzero`. You should be able to get `make grade` to succeed on the `divzero`, `softint`, and `badsegment` tests at this point.

完成这个 Exercise 主要需要完成两项工作：一是在 kern/trapentry.S 中定义各中断对应的中断处理程序（同时需要完成_alltraps 部分的汇编代码），二是在 kern/trap.c

的 idt_init 函数中将上述中断处理程序装进 IDT 中。

对于第一项工作，JOS 提供了两个宏 TRAPHANDLER_NOEC 和 TRAPHANDLER 来简化中断处理程序的声明，这两个宏前者针对系统不放入错误码的中断或异常，后者针对系统会放入错误码的中断或异常。至于中断或者异常是否需要系统放入错误码，参考课设资料“Chapter5.pdf”，里面讲得很详细了。除了利用宏声明，还需要根据 Exercise 的提示完成“_alltraps”部分代码，如图 3.7 所示。

```
71 _alltraps:
72     pushw   $0
73     pushw   %ds
74     pushw   $0
75     pushw   %es
76     pushal
77
78     movl    $GD_KD, %eax
79     movw    %ax, %ds
80     movw    %ax, %es
81
82     pushl   %esp
83
84     call    trap
```

图 3.7 _alltraps 代码

对于第二项工作，JOS 也提供了一个宏 SETGATE 用来设置特定的描述符，由于这部分代码重复性太多，所以不贴图了。需要注意的地方有两处，一是 SETGATE 第三个参数 cs 应设置为内核的代码段 GD_KT；二是 SETGATE 最后一个参数权限的设置，应设置为 0（内核），但是断点（BreakPoint）中断和 syscall 除外，应设置为用户态即设置权限参数为 3。

Questions：

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e.， if all exceptions/interrupts were delivered to the same handler， what feature that exists in the current implementation could not be provided?)

2. Did you have to do anything to make the user/softint program behave correctly (i.e.，as the grade script expects)? Why is this the correct behavior? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

1. JOS 的中断处理程序在真正的处理之前要将中断号放入内核栈以组织成

Trapframe 的结构，但是如果所有中断都跳到同一个处理程序，那么就无法区分是哪个中断调用进来的，也就无法正确设置它们的中断号了。

2. 目前 IDT 中 14 号中断 Page Fault 的权限为 0，只能由内核触发，所以如果直接在 softint 中用 int 指令调用会产生“General Protection Fault”的权限错误。如果修改 14 号中断 Page Fault 的权限为 3，则用户可以触发，便会给出“Page Fault”的异常。

3.2 Page Faults, Breakpoints Exceptions, and System Calls

3.2.1 Handling Page Faults

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them.

这个 Exercise 极为简单，只需在 kern/trap.c 的 trap_dispatch 函数开始位置添加如下代码即可：

```
if (tf->tf_trapno == T_PGFLT) {
    page_fault_handler(tf);
    return;
}
```

3.2.2 The Breakpoint Exception

Exercise 6. Modify `trap_dispatch()` to make breakpoint exceptions invoke the kernel monitor. You should now be able to get `make grade` to succeed on the `breakpoint` test.

与上个 Exercise 一样，在同样位置再添加如下代码即可：

```
if (tf->tf_trapno == T_BRKPT) {
    monitor(tf);
    return;
}
```

Questions

1. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e.,

your call to SETGATE from idt_init). Why? How did you need to set it in order to get the breakpoint exception to work as specified above?

2. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

1. 需要将 breakpoint 异常的权限设置为 3（用户级）来保证用户也可以触发改异常。
2. 安全

3.2.3 System calls

Exercise 7. Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `idt_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read `inc/syscall.h`.

Run the `user/hello` program under your kernel. It should print "`hello, world`" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right.

这个 Exercise 开始对 syscall 作处理。

首先，syscall 作为一种特殊的异常，也需要在 kern/trapentray.S 和 kern/trap.c 的 idt_init 函数添加相关的中断服务程序的声明和中断向量的装入的代码。

然后，修改 kern/trap.c 中的 dispatch 函数，添加 syscall 相关的处理代码。如图 3.8 所示，需要注意的是 syscall 函数的返回值被放到了寄存器%eax 中。

```
167     if (tf->tf_trapno == T_SYSCALL) {
168         int ret;
169         ret = syscall(
170             tf->tf_regs.reg_eax,
171             tf->tf_regs.reg_edx,
172             tf->tf_regs.reg_ecx,
173             tf->tf_regs.reg_ebx,
174             tf->tf_regs.reg_edi,
175             tf->tf_regs.reg_esi
176             );
177         if (ret < 0)
178             panic("trap dispatch: System call number is invalid. %e", ret);
179         tf->tf_regs.reg_eax = ret;
180         return;
181     }
```

图 3.8 dispatch 中关于 syscall 的代码

最后，完成 kern/syscall.c 中的 syscall 函数。该 syscall 其实相当于是一个分发函数，根据传入的 syscall 的编号，也就是形参 syscallno，调用对应的内核函数，其代码如图 3.9 所示。

```
77 int32_t
78 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
79 {
80     // Call the function corresponding to the 'syscallno' parameter.
81     // Return any appropriate return value.
82     // LAB 3: Your code here.
83
84     int ret = 0;
85     switch(syscallno) {
86         case SYS_cputs:
87             sys_cputs((const char *)a1, (size_t)a2);
88             break;
89         case SYS_cgetc:
90             ret = sys_cgetc();
91             break;
92         case SYS_getenvid:
93             ret = sys_getenvid();
94             break;
95         case SYS_env_destroy:
96             ret = sys_env_destroy((envid_t)a1);
97             break;
98         default:
99             ret = -E_INVAL;
100    }
101    return ret;
102 }
```

图 3.9 syscall 函数

3.2.4 User-mode startup

Exercise 8. Add the required code to the user library, then boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00000800`". `user/hello` then attempts to "exit" by calling `sys_env_destroy()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor.

这个 Exercise 的目的是为了设置全局变量 env 指向 envs 数组中当前进程的结构体，获取当前进程的 id 需要利用内核函数 sys_getenvid。按照前面的提示，在 lib/libmain.c 的 libmain 函数的开始位置，添加下面代码：

env = envs + ENVX(sys_getenvid());

3.2.5 Page faults and memory protection

Exercise 9. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

Change `kern/init.c` to run `user/buggyhello` instead of `user/hello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should *not* panic. You should see:

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

这个 Exercise 的要求很详细，没什么特别需要注意的，按照要求完成即可。

首先是修改 kern/trap.c 中 page_fault_handler 函数，添加判断异常来源是否为内核的代码，如下：

```
if ((tf->tf_cs & 3) == 0)
```

panic("Page fault in kernel mode");

然后，完成检测用户级页面是否有效的函数 user_mem_check，代码如图 3.10。

```
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    uintptr_t lva = (uintptr_t)va;
    uintptr_t hva = (uintptr_t)(va + len);
    uintptr_t iva ;
    perm = perm | PTE_U | PTE_P;
    for (iva = lva; iva < hva; iva += PGSIZE) {
        if (iva < ULIM) {
            pte_t *pte = pgdir_walk(env->env_pgdir, (void *)iva, 0);
            if (!pte || ((*pte & perm) != perm)) {
                user_mem_check_addr = iva;
                return -E_FAULT;
            }
        }
        else {
            user_mem_check_addr = iva;
            return -E_FAULT;
        }
        iva = ROUNDDOWN(iva, PGSIZE);
    }
    return 0;
}
```

图 3.10 user_mem_check 函数

最后，在 kern/syscall.c 的 sys_cputs 函数以及 kern/kdebug.c 的 debuginfo_eip 函数中添加对相关地址的有效性检查即可。

Exercise 10.

Change `kern/init.c` to run `user/evilhello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should not panic. You should see:

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

这个 Exercise 基本就是为了检查，可以跳过。

至此，Lab 3 的所有 Exercise 都已完成。完成后， 执行“sh grade.sh”进行评分，结果如图 3.11 图 2.21 所示。注意到，“testbss”这项测试并没有通过。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab3]
└─► $ sh grade.sh
hello: OK (s)
buggyhello: OK (s)
evilhello: OK (s)
divzero: OK (s)
breakpoint: OK (s)
softint: OK (s)
badsegment: OK (s)
faultread: OK (s)
faultreadkernel: OK (s)
faultwrite: OK (s)
faultwritekernel: OK (s)
testbss: missing 'Making sure bss works right...'
missing 'Yes, good.  Now doing a wild write off the end...'
missing '.00001000. user fault va 00c..... ip 008.....'
missing '.00001000. free env 00001000'
WRONG (s)
Score: 55/60
┌─[✗]─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab3]
└─► $
```

图 3.11 评分测试程序未通过

但问题是，在 MIT 的课程网页里并没有提到 testbss 这项测试的内容和应该通过的时间节点，所以很不容易定位出错位置，只能一点点的测试和找 bug 了。

首先，看了下 user/testbss.c 文件中的内容，并没有获取什么有价值的信息。然后，联想到其文件名为“testbss”，所以应该测试的是 BSS 节，因此，使用 objdump 查看 ELF 中的分段信息，如图 3.12 所示。注意到有一个加载段的 memsz 极其的大，其大小为 0x400028。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab3]
└─► $ objdump -x obj/user/testbss

obj/user/testbss:     file format elf32-i386
obj/user/testbss
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00800020

Program Header:
    LOAD off    0x00001000 vaddr 0x00200000 paddr 0x00200000 align 2**12
         filesz 0x00003cfd memsz 0x00003cfd flags rw-
    LOAD off    0x00005020 vaddr 0x00800020 paddr 0x00800020 align 2**12
         filesz 0x00001207 memsz 0x00001207 flags r-x
    LOAD off    0x00007000 vaddr 0x00802000 paddr 0x00802000 align 2**12
         filesz 0x00000008 memsz 0x00400028 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

图 3.12 testbss 的 ELF 信息

单独执行 testbss 程序（修改 init 函数），运行截图如图 3.13 所示。

```
========================================================================
                    Bochs x86 Emulator 2.4.6
            Build from CVS snapshot, on February 22, 2011
                  Compiled at Feb  6 2017, 00:14:56
========================================================================
00000000000i[     ] reading configuration from .bochsrc
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> c
Physical memory: 32768K available, base = 640K, extended = 31744K
check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
[00000000] new env 00001000
kernel panic at kern/env.c:217: segment_alloc: Allocate physical page failed.

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 3.13 testbss 运行结果

发现 segment_alloc 函数发生了异常，然后去查看 segment_alloc 函数，仔仔细细的思考后，我认为，造成异常的最可能的原因是内存空间不够。但为什么内存空间会不够呢？

找了大约一天的原因后，我突然意识到有可能是 page_init 函数里分配的物理页太少了。返回去看 kern/pmap.c 文件中的 page_init 函数，发现果然是原来的实现有

点问题。修改后，重新编译执行评分测试程序，结果如图 3.14 所示。得到的评分为“60/60”，终于通过了 Lab 3。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab3]
└──╼ $ sh grade.sh
hello: OK (s)
buggyhello: OK (s)
evilhello: OK (s)
divzero: OK (s)
breakpoint: OK (s)
softint: OK (s)
badsegment: OK (s)
faultread: OK (s)
faultreadkernel: OK (s)
faultwrite: OK (s)
faultwritekernel: OK (s)
testbss: OK (s)
Score: 60/60
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab3]
└──╼ $
```

图 3.14 评分测试程序（通过）

4 Preemptive Multitasking

4.1 User-level Environment Creation and Cooperative Multitasking

4.1.1 Round-Robin Scheduling

Exercise 1. Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Modify `kern/init.c` to create two (or more!) environments that all run the program `user/yield.c`. You should see the environments switch back and forth between each other five times before terminating, like this:

```
...
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

After the `yield` programs exit, the idle environment should run and invoke the JOS kernel debugger. If all this does not happen, then fix your code before proceeding.

这个 Exercise 的目的是实现轮转循环调度，其关键是 kern/sched.c 中的 sched_yield 函数，按照注释实现其代码如图 4.1 所示。sched_yield 函数的关键在于第 18-26 行的 for 循环，该循环从 envs 中（以当前 env 的下一个 env 作为起点）寻找一个可以运行的 env，需要注意的是第 19-21 行，如果寻找到了 envs 数组的最后一个 env，下一个应该寻找 envs[1]（不是 envs[0]）。如果 envs 数组中没有可以执行的 env，再考虑 envs[0]。

```
 9 void
10 sched_yield(void)
11 {
12     int i;
13     struct Env *ienv;
14     if (curenv == NULL)
15         ienv = envs + 1;
16     else
17         ienv = curenv + 1;
18     for (i = 0 ; i < NENV; i++, ienv++) {
19         if (ienv >= envs + NENV) {
20             ienv = envs + 1;
21         }
22         if (ienv->env_status == ENV_RUNNABLE) {
23             env_run(ienv);
24             return;
25         }
26     }
27
28     // Run the special idle environment when nothing else is runnable.
29     if (envs[0].env_status == ENV_RUNNABLE) {
30         env_run(&envs[0]);
31     }
32     else {
33         cprintf("Destroyed all environments - nothing more to do!\n");
34         while (1)
35             monitor(NULL);
36     }
37 }
```

图 4.1 sched_yield 函数

除此之外，根据 Exercise 的要求，还需要修改 kern/syscall.c 中的 syscall 函数添

加相关的分配机制，修改 kern/init.c 中的 i386_init 函数使其运行一个 user_idle 和两个 user_yield，修改完毕后重新编译运行 Bochs，运行截图如图 4.2 所示。

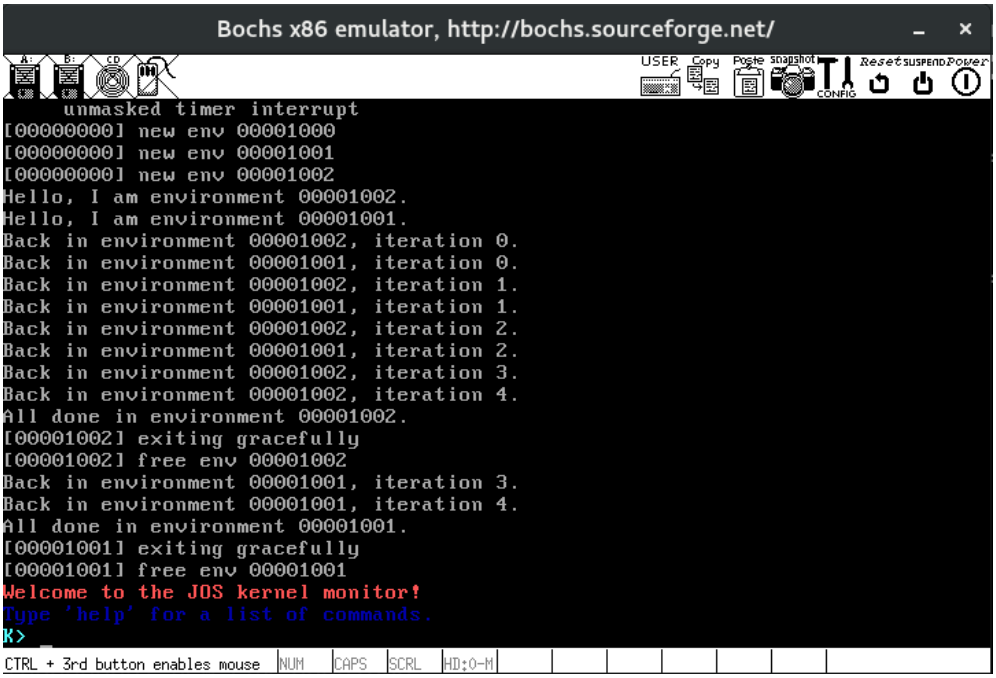

图 4.2 Exercise 1 运行截图

Question:

In your implementation of env_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

e 的地址实际上是在 KERNBASE 后面的高地址区，对于所有的用户和内核地址空间而言，这片地址区指向同样的物理内存空间，所以切换页表前后 e 不受影响。

4.1.2 System Calls for Environment Creation

Exercise 2. Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.cc`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

这个部分最重要的是理解 sys_exofork 这个函数，关于父子进程调用该函数的返回值的区别等等。由于上学期做操作系统实验的时候学习和使用过 fork 函数，所以这里理解起来相对容易一些。

首先考虑刚提到的 sys_exofork 函数，按照注释的要求，该函数需要使用 env_alloc 函数创建一个新的进程，然后赋值父进程的寄存器的内容，并设置子进程返回值为 0（第 95 行，关键），父进程返回子进程 envid，具体的代码如图 4.3 所示。

```
85 static envid_t
86 sys_exofork(void)
87 {
88     // LAB 4: Your code here.
89     int err;
90     struct Env *env;
91     err = env_alloc(&env, curenv->env_id);
92     if (err >= 0) {
93         env->env_status = ENV_NOT_RUNNABLE;
94         env->env_tf = curenv->env_tf;
95         env->env_tf.tf_regs.reg_eax = 0;
96         return env->env_id;
97     }
98     else
99         return err;
100
101 }
```

图 4.3 sys_exofork 函数

然后考虑 sys_env_set_status 函数，这个函数相当简单，唯一需要注意的地方是用到了 JOS 已经实现的 envid2env 函数，具体的代码如图 4.4 所示。

```
116 static int
117 sys_env_set_status(envid_t envid, int status)
118 {
119     // LAB 4: Your code here.
120     if (status == ENV_RUNNABLE || status == ENV_NOT_RUNNABLE) {
121         struct Env *env = NULL;
122         int err = envid2env(envid, &env, 1);
123         if (err == 0) {
124             env->env_status = status;
125             return 0;
126         }
127         else
128             return err;
129     }
130     else
131         return -E_INVAL;
132 }
```

图 4.4 sys_env_set_status 函数

接着考虑 sys_page_alloc 函数，函数本身其实并不难，主要是利用 page_insert 函数在 env 的指定虚地址处绑定一个物理页，需要注意的地方是课设资料中提到了最好将新申请的物理页全部清零。但麻烦的是需要异常检测和判断，有各种各样的错误情形，仔细按照注释要求实现即可，具体的函数代码如图 4.5 所示。

然后考虑 sys_page_map 函数，与上个函数很相似，只是不需要申请物理页的分配了，不多赘述，具体的代码如图 4.6 所示。

最后考虑 sys_page_unmap 函数，实现起来也是非常简单，只需要调用下之前

```
189 static int
190 sys_page_alloc(envid_t envid, void *va, int perm)
191 {
192     // LAB 4: Your code here.
193     int err;
194     struct Env *env;
195     if (va >= (void *)UTOP)
196         return -E_INVAL;
197     if ((perm & PTE_U) == 0 && (perm & PTE_P) == 0)
198         return -E_INVAL;
199     if ((perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0)
200         return -E_INVAL;
201     err = envid2env(envid, &env, 1);
202     if (err == 0) {
203         struct Page *page;
204         err = page_alloc(&page);
205         if (err == 0) {
206             err = page_insert(env->env_pgdir, page, va, perm);
207             if (err == 0) {
208                 memset(page2kva(page), 0, PGSIZE);
209                 return 0;
210             }
211             else {
212                 page_free(page);
213                 return err;
214             }
215         }
216         else
217             return err;
218     }
219     else
220         return err;
221 }
```

图 4.5 sys_page_alloc 函数

```
254 static int
255 sys_page_map(envid_t srcenvid, void *srcva,
256              envid_t dstenvid, void *dstva, int perm)
257 {
258     // LAB 4: Your code here.
259     struct Env *srcenv, *dstenv;
260     int err;
261     pte_t *pte;
262     struct Page *page;
263     if (srcva >= (void *)UTOP || srcva != ROUNDUP(srcva, PGSIZE))
264         return -E_INVAL;
265     if (dstva >= (void *)UTOP || dstva != ROUNDUP(dstva, PGSIZE))
266         return -E_INVAL;
267     if ((perm & PTE_U) == 0 && (perm & PTE_P) == 0)
268         return -E_INVAL;
269     if ((perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0)
270         return -E_INVAL;
271     err = envid2env(srcenvid, &srcenv, 1);
272     if (err < 0)
273         return err;
274     err = envid2env(dstenvid, &dstenv, 1);
275     if (err < 0)
276         return err;
277     page = page_lookup(srcenv->env_pgdir, srcva, &pte);
278     if (page == NULL || ((perm & PTE_W) != 0 && (*pte & PTE_W) == 0))
279         return -E_INVAL;
280     err = page_insert(dstenv->env_pgdir, page, dstva, perm);
281     if (err < 0)
282         return err;
283     return 0;
284 }
```

图 4.6 sys_page_map 函数

实现的 page_remove 函数即可，具体的代码如图 4.7 所示。至此，本 Exercise 通过。

```
293 static int
294 sys_page_unmap(envid_t envid, void *va)
295 {
296     // Hint: This function is a wrapper around page_remove().
297
298     // LAB 4: Your code here.
299     int err;
300     struct Env *env;
301     if (va >= (void *)UTOP || va != ROUNDUP(va, PGSIZE))
302         return -E_INVAL;
303     err = envid2env(envid, &env, 1);
304     if (err == 0) {
305         page_remove(env->env_pgdir, va);
306         return 0;
307     }
308     else
309         return err;
310 }
```

图 4.7 sys_page_unmap 函数

4.2 Copy-on-Write Fork

4.2.1 User-level page fault handling

这一部分不用实现什么，看 MIT 的课程网站即可。

4.2.2 Setting the Page Fault Handler

Exercise 4. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

实现真的很简单，代码如图 4.8 所示。

```
164 static int
165 sys_env_set_pgfault_upcall(envid_t envid, void *func)
166 {
167     // LAB 4: Your code here.
168     int err;
169     struct Env *env;
170     err = envid2env(envid, &env, 1);
171     if (err == 0) {
172         env->env_pgfault_upcall = func;
173         return 0;
174     }
175     else
176         return err;
177 }
```

图 4.8 sys_env_set_pgfault_upcall 函数

4.2.3 Normal and Exception Stacks in User Environments

重新阅读 inc/memlayout.h 中的内存布局，尤其是栈的部分。一共有三个栈，分别是：内核系统栈[KSTACKTOP, KSTACKTOP − KSTKSIZE)，一般用户栈[USTACKTOP, UTEXT)，用户异常栈[UXSTACKTOP, UXSTACKTOP − PGSIZE)。

当用户态 env 正在运行时，所在的栈为其自身的一般用户栈；当中断或异常来临时，系统陷入内陷状态，栈也会从一般用户栈切换到内核系统栈；对于用户级的缺页异常处理程序，栈会由系统栈切换到用户异常栈；返回用户程序时，栈切换回一般用户栈。

4.2.4 Invoking the User Page Fault Handler

Exercise 5. Implement the code in `kern/trap.c` required to dispatch page faults the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

这一部分需要完成的是 kern/trap.c 中的 page_falut_handler 函数（Lab 3 已经完

成一部分，Lab 4 需要完成用户态中发生的页错误的处理），该函数需要将触发页错误的进程信息以 Utrapframe（定义在 inc/trap.h 中，需要仔细研究）结构体的形式压入用户异常栈中，然后调用用户的缺页异常处理程序。

这里需要注意的地方是，用户态下造成缺页中断的情景有两类：一是用户程序运行中访问到错误地址，触发了缺页异常；二是用户程序自定义的缺页异常处理程序在运行时触发了缺页异常。所以，极有可能造成用户自定义的缺页异常处理程序的递归调用。

具体的代码（添加的部分）如图 4.9 所示，第 314 行是判断触发缺页异常的情景是前面哪一种，第 315 和 317 行是两种情形下设置的用户异常栈栈顶位置（正好是 1 个 Utrapframe 结构体），注意 315 行预留了 4 个字节（用处后面会提到）。第 319 到 324 行是赋值 tf 中的内容到 utf，第 325 行切换栈到用户异常栈，第 326 行设置%eip 为用户自定义的缺页异常处理程序。

```
312     if (curenv->env_pgfault_upcall != NULL) {
313         struct UTrapframe *utf;
314         if (tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP - PGSIZE)
315             utf = (struct UTrapframe *)(tf->tf_esp - 4 - sizeof(struct UTrapframe));
316         else
317             utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct UTrapframe));
318         user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_U | PTE_W);
319         utf->utf_fault_va = fault_va;
320         utf->utf_err = tf->tf_err;
321         utf->utf_regs = tf->tf_regs;
322         utf->utf_eip = tf->tf_eip;
323         utf->utf_eflags = tf->tf_eflags;
324         utf->utf_esp = tf->tf_esp;
325         curenv->env_tf.tf_esp = (uint32_t)utf;
326         curenv->env_tf.tf_eip = (uint32_t)curenv->env_pgfault_upcall;
327         env_run(curenv);
328     }
```

图 4.9 page_falut_handler 函数添加部分

4.2.5 User-mode Page Fault Entrypoint

Exercise 6. Implement the `_pgfault_upcall` routine in `lib/pfentry.S`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

_pgfault_upcall 是所有用户自定义缺页异常处理程序的入口地址，其会调用用户自定义缺页异常处理程序，并在处理程序执行完毕后从用户异常栈恢复相应信息，并跳转到用户程序继续执行。

_pgfault_upcall 的实现极其巧妙，自己并没有独立思考出来，参考了网上的相关资料和学长的代码后才体会到这段汇编代码的神奇，如图 4.10 所示。这段代码的

注释是自己经过反复阅读这段代码和思考后写上的（习惯于英文注释）。

```
68      // Code below is to modify the trap-time esp (store in UTrapframe) to trap-tiem esp - 4
69      movl    0x30(%esp), %eax
70      subl    $0x4, %eax
71      movl    %eax, 0x30(%esp)
72      // Code below is to store trap-time eip to trap-time esp
73      movl    0x28(%esp), %ebx
74      movl    %ebx, (%eax)
75
76      // Restore the trap-time registers.
77      // LAB 4: Your code here.
78
79      // Skip va and error code
80      addl    $0x8, %esp
81      popal
82
83      // Restore eflags from the stack.
84      // LAB 4: Your code here.
85
86      // Skip eip
87      addl    $0x4, %esp
88      popfl
89
90      // Switch back to the adjusted trap-time stack.
91      // LAB 4: Your code here.
92
93      pop     %esp
94
95      // Return to re-execute the instruction that faulted.
96      // LAB 4: Your code here.
97
98      ret
```

图 4.10 _pgfault_upcall 实现

通过注释，可以理解大部分代码的意图。需要注意的是当分别执行完“popal”和“popfl”后，就不能再使用所有的通用寄存器和标志寄存器。除此之外，前面提到过在用户自定义缺页异常处理程序递归调用时，会在用户异常栈为其预留 4 字节的空间，这 4 字节的空间就是为了放置原出错序的 EIP，以便最后 ret 时正好将这 4 个字节的内容写回到寄存器%eip 中。

Exercise 7. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

这个 Exercise 跟上一个相比简单很多，set_pgfault_handler 函数主要是申请用户异常栈的内存空间，并且设置自定义的缺页异常处理程序，具体的代码如图 4.11 所示。

```
22  void
23  set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
24  {
25      int r;
26
27      // First time through!
28      if (_pgfault_handler == 0) {
29          r = sys_page_alloc(0, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W);
30          if (r < 0)
31              panic("set_pgfault_handler: Set failed. %e", r);
32          else
33              sys_env_set_pgfault_upcall(0, _pgfault_upcall);
34      }
35
36      // Save handler pointer for assembly to call.
37      _pgfault_handler = handler;
38  }
```

图 4.11 set_pgfault_handler 函数

4.2.6 Implementing Copy-on-Write Fork

Exercise 8. Implement `fork` and `pgfault` in `lib/fork.c`.

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1001: I am ''
1802: I am '0'
2801: I am '00'
3802: I am '000'
2003: I am '1'
5001: I am '11'
4802: I am '10'
6801: I am '100'
5803: I am '110'
3004: I am '01'
8001: I am '011'
7803: I am '010'
4005: I am '001'
6006: I am '111'
7007: I am '101'
```

这个 Exercise 的难度也挺大的，完成之前一定要认真阅读 MIT 的课程网站。

首先考虑 pgfault 函数，这个函数中需要用户程序访问页目录和页表，这就要涉及到 VPT、UVPT、vpt 和 upd（在 lib/entry.S 中有相应的设置）。假设查询的虚地址为 addr = PDX| PTX| OFFSET，若要得到对应页目录表项，则访问虚地址 vaddr = UVPT[31:22]| UVPT[31:22]| PDX| 00；若要得到对应页表表项，则访问虚地址 vaddr = UVPT[31:22]| PDX| PTX| 00。这个关系很难讲清楚，但是一定要自己推算。在用户程序中，则可以利用 vpt 和 vpd 访问页目录和页表，假设需要查询的虚拟地址为 va，则其对应的页目录表项为：vpd[VPD(va)]，其中宏 VPD 等价于宏 PDX；则其对应的页表表象为：vpt[VPN(va)]，其中宏 VPN 等价于 PPN。

有了上面的介绍，按照注释，pgfault 函数的实现不难完成，如图 4.12 所示。

```
14 static void
15 pgfault(struct UTrapframe *utf)
16 {
17     void *addr = (void *) utf->utf_fault_va;
18     uint32_t err = utf->utf_err;
19     int r;
20
21     // Check that the faulting access was (1) a write, and (2) to a
22     // copy-on-write page.  If not, panic.
23     // Hint:
24     //   Use the read-only page table mappings at vpt
25     //   (see <inc/memlayout.h>).
26
27     // LAB 4: Your code here.
28
29     if ((err & FEC_WR) == 0 || (vpd[VPD(addr)] & PTE_P) == 0 || (vpt[VPN(addr)] & PTE_COW) == 0)
30         panic("pgfault: the faulting access was not a write or to a copy-on-write page");
31
32     // Allocate a new page, map it at a temporary location (PFTEMP),
33     // copy the data from the old page to the new page, then move the new
34     // page to the old page's address.
35     // Hint:
36     //   You should make three system calls.
37     //   No need to explicitly delete the old page's mapping.
38
39     // LAB 4: Your code here.
40
41     r = sys_page_alloc(0, (void *)PFTEMP, PTE_U | PTE_W | PTE_P);
42     if (r < 0)
43         panic("pgfault: new page allocate failed");
44     addr = ROUNDDOWN(addr, PGSIZE);
45     memmove(PFTEMP, addr, PGSIZE);
46     r = sys_page_map(0, (void *)PFTEMP, 0, addr, PTE_U | PTE_W | PTE_P);
47     if (r < 0)
48         panic("pgfault: page map failed");
49 }
```

图 4.12 pgfault 函数

然后考虑 duppage 函数，该函数用于进行 COW 方式的页复制，唯一需要注意的地方是其传入的参数为物理页号，应进行相应的地址运算，其代码如图 4.13 所示。

```
61 static int
62 duppage(envid_t envid, unsigned pn)
63 {
64     int r;
65     void *addr;
66     pte_t pte;
67
68     // LAB 4: Your code here.
69
70     addr = (void *)(pn * PGSIZE);
71     pte = vpt[VPN(addr)];
72     if ((pte & PTE_W) != 0 || (pte & PTE_COW) != 0) {
73         r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_COW | PTE_P);
74         if (r >= 0) {
75             r = sys_page_map(0, addr, 0, addr, PTE_U | PTE_COW | PTE_P);
76             if (r < 0)
77                 return r;
78             else
79                 return 0;
80         }
81         else
82             return r;
83
84     }
85     else {
86         r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_P);
87         if (r < 0)
88             return r;
89         else
90             return 0;
91     }
92 }
```

图 4.13 duppage 函数

最后是 fork 函数，需要注意的地方有：第 116 行调用的是 set_pgfault_handler 而不是系统调用，因为前者会检查是否为用户程序分配了用户异常栈；第 121 行的 for 循环，是将父进程 0 到 UXSTACKTOP 之间的所有用户页面以 COW 的方式复制给子进程；注意所有可能的异常处理（返回值）。

```
110 envid_t
111 fork(void)
112 {
113     int r;
114     uint32_t addr;
115     envid_t envid;
116     set_pgfault_handler(pgfault);
117     envid = sys_exofork();
118     if (envid >= 0) {
119         // Parent
120         if (envid > 0) {
121             for (addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE) {
122                 if ((vpd[VPD(addr)] & PTE_P) != 0 && (vpt[VPN(addr)] & PTE_P) != 0 && (vpt[VPN(addr)] & PTE_U) != 0)
123                     duppage(envid, VPN(addr));
124             }
125             r = sys_page_alloc(envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
126             if (r == 0) {
127                 extern void _pgfault_upcall(void);
128                 r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
129                 if (r == 0) {
130                     r = sys_env_set_status(envid, ENV_RUNNABLE);
131                     if (r == 0)
132                         return envid;
133                     else
134                         return 0;
135                 }
136                 else
137                     return r;
138             }
139             else
140                 return r;
141         }
142         // Child
143         else {
144             env = envs + ENVX(sys_getenvid());
145             return 0;
146         }
147     }
148     // Error
149     else
150         return envid;
151 }
```

图 4.14 fork 函数

完成了 fork 函数，这个 Exercise 和 Part 也就完成了，可以说这一部分是做课设以来遇到的最大挑战，真的很难。

4.3 Preemptive Multitasking and Inter-Process communication (IPC)

4.3.1 Interrupt discipline

Exercise 9. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `dumbfork`), you should see a kernel panic shortly into the program's execution. This is because our code has set up the clock hardware to generate clock interrupts, and interrupts are now enabled in the processor, but JOS isn't yet handling them.

该部分参考 Lab 3 中定义异常处理程序和设置 IDT 的部分即可。

注意对于中断系统是不会压入错误码的，所以利用宏 TRAPHANDLER_NOEC 即可；对于所有的中断，应该由系统产生，所以其权限应该设置为 0。

然后，还需要修改 kern/env.c 中的 env_alloc 函数，在创建用户进程时，将其 EFLAGS 寄存器中的 IF 为置为 1 即可，所以在 env_alloc 函数中加上一句代码：

e->env_tf.tf_eflags |= FL_IF;

4.3.2 Handling Clock Interrupts

Exercise 10. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

这个 Exercise 还是十分简单的，只需在 trap_dispatch 的开始位置加上对时钟中断的处理（调用 sched_yield 轮转调度函数即可）即可，时钟中断的中断号可以通过两个宏 IRQ_OFFSET 和 IRQ_TIMER 得到。这样，当每个时钟周期到来时，当前 env 都需要让出 CPU，具体添加的代码如图 4.15 所示。

```
222     // Handle clock and serial interrupts.
223     // LAB 4: Your code here.
224
225     if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
226         sched_yield();
227         return;
228     }
```

图 4.15 时钟中断处理

4.3.3 Implementing IPC

Exercise 11. Implement `sys_ipc_recv` and `sys_ipc_can_send` in `kern/syscall.c`. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

这个 Exercise 虽然需要写的代码不少，但因为注释很详细，所以相对都比较好写。首先，需要添加两个 syscall：sys_ipc_try_send 和 sys_ipc_recv，并在 syscall 函数中添加对应的分发机制，其代码分别如图 4.16 和图 4.17 所示。

```
353 static int
354 sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
355 {
356     // LAB 4: Your code here.
357     struct Env *env;
358     int err;
359     pte_t *pte;
360     struct Page *page;
361     //  -E_BAD_ENV if environment envid doesn't currently exist.
362     err = envid2env(envid, &env, 0);
363     if (err < 0)
364         return err;
365     //  -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
366     //      or another environment managed to send first.
367     if (env->env_ipc_recving != 1 || env->env_ipc_from != 0)
368         return -E_IPC_NOT_RECV;
369     if (srcva < (void *)UTOP) {
370         err = -E_INVAL;
371         if (ROUNDDOWN(srcva, PGSIZE) != srcva)
372             return err;
373         if ((perm & PTE_U) == 0 && (perm & PTE_P) == 0)
374             return err;
375         if ((perm & ~(PTE_U | PTE_P | PTE_AVAIL | PTE_W)) != 0)
376             return err;
377         page = page_lookup(curenv->env_pgdir, srcva, &pte);
378         if (page == NULL)
379             return err;
380         if ((perm & PTE_W) != 0 && (*pte & PTE_W) == 0)
381             return err;
382         if (env->env_ipc_dstva < (void *)UTOP) {
383             err = page_insert(env->env_pgdir, page, env->env_ipc_dstva, perm);
384             if (err < 0)
385                 return err;
386             else
387                 env->env_ipc_perm = perm;
388         }
389     }
390     env->env_ipc_recving = 0;
391     env->env_ipc_from = curenv->env_id;
392     env->env_ipc_value = value;
393
394     env->env_status = ENV_RUNNABLE;
395     env->env_tf.tf_regs.reg_eax = 0;
396
397     return 0;
398 }
```

图 4.16 sys_ipc_try_send 函数

```
411 static int
412 sys_ipc_recv(void *dstva)
413 {
414     // LAB 4: Your code here.
415     if (dstva < (void *)UTOP && ROUNDDOWN(dstva, PGSIZE) != dstva) {
416         return -E_INVAL;
417     }
418     curenv->env_ipc_recving = 1;
419     curenv->env_ipc_dstva = dstva;
420     curenv->env_status = ENV_NOT_RUNNABLE;
421     curenv->env_ipc_from = 0;
422
423     sched_yield();
424 }
```

图 4.17 sys_ipc_recv 函数

然后实现两个库函数 ipc_recv 和 ipc_send 函数，分别如图 4.18 和图 4.19 所示。

```
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    int err;
    if (pg != NULL)
        err = sys_ipc_recv((void *)UTOP);
    else
        err = sys_ipc_recv(pg);
    if (from_env_store != NULL) {
        if (err < 0)
            *from_env_store = 0;
        else
            //*from_env_store = env->env_ipc_from;
            *from_env_store = envs[ENVX(sys_getenvid())].env_ipc_from;
    }
    if (perm_store != NULL) {
        if (err < 0)
            *perm_store = 0;
        else
            //*perm_store = env->env_ipc_perm;
            *perm_store = envs[ENVX(sys_getenvid())].env_ipc_perm;

    }
    if (err < 0)
        return err;
    else
        return env->env_ipc_value;
}
```

图 4.18 ipc_recv 函数

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    int err;
    /*
    *do {
    *    if (pg != NULL)
    *        err = sys_ipc_try_send(to_env, val, pg, perm);
    *    else
    *        err = sys_ipc_try_send(to_env, val, (void *)UTOP, perm);
    *    cprintf("%d\n", err);
    *    if (err != -E_IPC_NOT_RECV) {
    *        panic("ipc_send: send message failed. %e", err);
    *    }
    *    sys_yield();
    *}
    *while (err < 0);
    */
    while ((err = sys_ipc_try_send(to_env, val, pg != NULL ? pg : (void *)UTOP, perm)) < 0) {
        if (err != -E_IPC_NOT_RECV)
            panic("ipc_send: send message failed. %e", err);
        sys_yield();
    }
}
```

图 4.19 ipc_send 函数

至此，Lab 4 的所有 Exercise 都已完成。完成后， 执行“sh grade.sh”进行评分，结果如图 4.20 所示。显然，PART C 的测试并没有通过，主要是 pingpong 和 primes。

```
┌─[dracula@localhost]─[~/Documents/IT_Study/MIT-JOS/code/lab4]
└──╼ $ sh grade.sh
dumbfork: OK (s)
PART A SCORE: 5/5
faultread: OK (s)
faultwrite: OK (s)
faultdie: OK (s)
faultalloc: OK (s)
faultallocbad: OK (s)
faultnostack: OK (s)
faultbadhandler: OK (s)
faultevilhandler: OK (s)
forktree: OK (s)
PART B SCORE: 50/50
spin: OK (s)
pingpong: missing '1002 got 0 from 1001'
missing '1001 got 1 from 1002'
missing '1002 got 8 from 1001'
missing '1001 got 9 from 1002'
missing '1002 got 10 from 1001'
missing '.00001001. exiting gracefully'
missing '.00001001. free env 00001001'
missing '.00001002. exiting gracefully'
missing '.00001002. free env 00001002'
WRONG (s)
primes: missing '2 .00001002. new env 00001003'
missing '3 .00001003. new env 00001004'
missing '5 .00001004. new env 00001005'
missing '7 .00001005. new env 00001006'
missing '11 .00001006. new env 00001007'
WRONG (s)
PART C SCORE: 55/65
```

图 4.20 Lab 4 评分测试程序（失败）

为了寻找错误的原因，单独运行和测试 pingpong 程序，执行“make run-pingpong”指令，重新编译和启动 Bochs，得到的运行结果如图 4.21 所示。从图中很容易看出，dispatch 函数给出的报错提示为 syscall 的编号为-7。

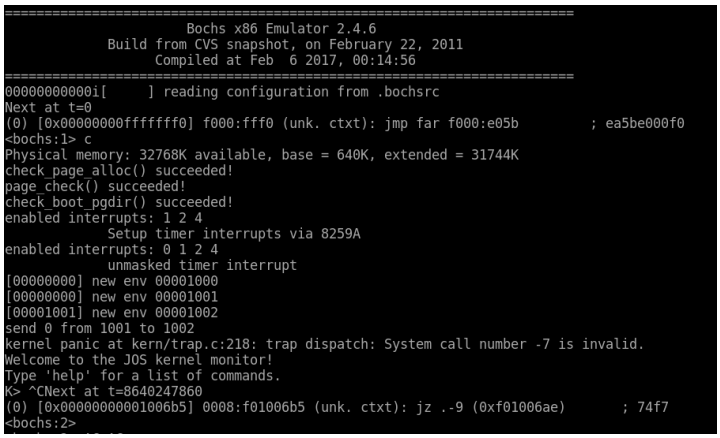

图 4.21 pingpong 测试程序运行结果

于是，寻找 dispatch 函数的相关 panic 函数，发现在前几个 Lab 中为了判断 syscall 的编号是否合法，用了一个判断来判断 syscall 的返回值 ret 是否小于 0。但是，sys_ipc_recv 函数的返回值本身就有可能小于 0（收到一个小于 0 的数据），所以会引发 panic。将该 if 判断注释掉，重新编译，执行“sh grade.sh”进行评分，结果如图 4.22 所示。显然，评分为满分，表示 Lab 4 已经通过。

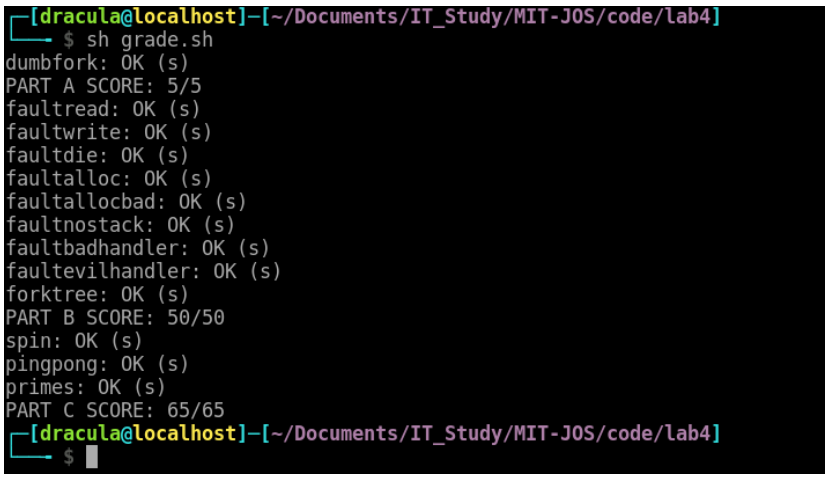

图 4.22 Lab 4 评分测试程序（通过）

5 File Systems and Spawn

5.1 The File System Server

5.1.1 Disk Access

> **Exercise 1.** Modify your kernel's environment initialization function, `env_alloc` in `env.c`, so that it gives environment 1 I/O privilege, but never gives that privilege to any other environment.
>
> Use `gmake grade` to test your code.

好像每个部分的第一个 Exercise 都比较简单，这个 Exercise 仿照 Lab 4 对 env_alloc 进行修改即可。主要在创建 envs[1]时，使其具有 IO 权限，添加的代码为：

```
if (e == envs + 1)
    e->env_tf.tf_eflags |= FL_IOPL_3;
```

5.1.2 The Block Cache

> **Exercise 2.** Implement the `read_block` and `write_block` functions in `fs/fs.c`. The `read_block` function should test to see if the requested block is already in memory, and if not, allocate a page and read in the block using `ide_read`. Keep in mind that there are multiple disk sectors per block/page, and that `read_block` needs to return the virtual address at which the requested block was mapped.
>
> The `write_block` function may assume that the indicated block is already in memory, and simply writes it out to disk. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the `PTE_D` "dirty" bit is set in the `vpt` entry. (The `PTE_D` bit is set by the processor; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block, `write_block` should clear the `PTE_D` bit using `sys_page_map`.
>
> Use `gmake grade` to test your code.

按照要求 MIT 的课程网站的要求和 JOS 的注释，完成 fs/fs.c 中的两个函数 read_block 和 write_block，两个函数主要是对 ide_read 和 ide_write 函数的封装，其代码分别如图 5.1 和图 5.2 所示。

```
65 static int
66 read_block(uint32_t blockno, char **blk)
67 {
68     int r;
69     char *addr;
70
71     if (super && blockno >= super->s_nblocks)
72         panic("reading non-existent block %08x\n", blockno);
73
74     if (bitmap && block_is_free(blockno))
75         panic("reading free block %08x\n", blockno);
76
77     // LAB 5: Your code here.
78     addr = diskaddr(blockno);
79     r = map_block(blockno);
80     if (r < 0)
81         return r;
82     r = ide_read(blockno * BLKSECTS, (void*)addr, BLKSECTS);
83     if (r < 0)
84         return r;
85     if (blk != NULL)
86         *blk = addr;
87     return 0;
88 }
```

图 5.1 read_block 函数

```
94 void
95 write_block(uint32_t blockno)
96 {
97     char *addr;
98
99     if (!block_is_mapped(blockno))
100        panic("write unmapped block %08x", blockno);
101
102    // Write the disk block and clear PTE_D.
103    // LAB 5: Your code here.
104    addr = diskaddr(blockno);
105    if (!block_is_dirty(blockno))
106        return;
107    if (ide_write(blockno * BLKSECTS, (void *)addr, BLKSECTS) < 0)
108        panic("write_block: IDE write failed.");
109    if (sys_page_map(0, (void *)addr, 0, (void *)addr, PTE_U|PTE_P|PTE_AVAIL|PTE_W) < 0)
110        panic("write_block: Syscall page map failed.");
111    return;
112 }
```

图 5.2 write_block 函数

5.1.3 The Block Bitmap

Exercise 3. Implement `read_bitmap`. It should check that all of the "reserved" blocks in the file system - block 0, block 1 (the superblock), and all the blocks holding the block bitmap itself, are marked in-use. Use the provided `block_is_free` routine for this purpose. You may simply panic if the file system is invalid.

Use `gmake grade` to test your code.

按要求实现 fs/fs.c 中的 read_bitmap 函数，如图 5.3 所示

```
221 void
222 read_bitmap(void)
223 {
224     int r, bitmap_blk_num;
225     uint32_t i;
226     char *blk;
227
228     // LAB 5: Your code here.
229
230     bitmap_blk_num = ROUNDUP(super->s_nblocks, BLKBITSIZE) / BLKBITSIZE;
231     for (i = 0; i < bitmap_blk_num; i++) {
232         r = read_block(2 + i, &blk);
233         if (r < 0)
234             panic("read_bitmap: Read block %08x failed.", 2 + i);
235     }
236     bitmap = (uint32_t *)diskaddr(2);
237
238     // Make sure the reserved and root blocks are marked in-use.
239     assert(!block_is_free(0));
240     assert(!block_is_free(1));
241     assert(bitmap);
242
243     // Make sure that the bitmap blocks are marked in-use.
244     // LAB 5: Your code here.
245
246     for (i = 0; i < bitmap_blk_num; i++)
247         assert(!block_is_free(2 + i));
248
249     cprintf("read_bitmap is good\n");
250 }
```

图 5.3 read_bitmap 函数

Exercise 4. Use `block_is_free` as a model to implement `alloc_block_num`, which scans the block bitmap for a free block, marks that block in-use, and returns the block number. When you allocate a block, you should *immediately* flush the changed bitmap block to disk with `write_block`, to help file system consistency.

Use `gmake grade` to test your code.

按要求实现 fs/fs.c 中的 alloc_block_num 函数，如图 5.4 所示。

```
156 int
157 alloc_block_num(void)
158 {
159     // LAB 5: Your code here.
160     int i;
161     for (i = 2 + ROUNDUP(super->s_nblocks, BLKSECTS)/BLKSECTS; i < super->s_nblocks; i++) {
162         if (block_is_free(i)) {
163             bitmap[i / 32] &= ~(1 << (i % 32));
164             write_block(2 + i / BLKBITSIZE);
165             return i;
166         }
167     }
168     return -E_NO_DISK;
169 }
```

图 5.4 alloc_block_num 函数

5.1.4 File Operations

> **Exercise 5.** Fill in the remaining functions in `fs/fs.c` that implement "top-level" file operations: `file_open`, `file_get_block`, `file_truncate_blocks`, and `file_flush`.
>
> Use `gmake grade` to test your code.

file_open 函数如图 5.5 所示。

```
576 int
577 file_open(const char *path, struct File **pf)
578 {
579     // Hint: Use walk_path.
580     // LAB 5: Your code here.
581     int r;
582     r = walk_path(path, NULL, pf, NULL);
583     if (r < 0)
584         return r;
585     else
586         return 0;
587 }
```

图 5.5 file_open 函数

file_get_block 函数如图 5.6 所示。

```
396 int
397 file_get_block(struct File *f, uint32_t filebno, char **blk)
398 {
399     int r;
400     uint32_t diskbno;
401
402     // Read in the block, leaving the pointer in *blk.
403     // Hint: Use file_map_block and read_block.
404     // LAB 5: Your code here.
405
406     r = file_map_block(f, filebno, &diskbno, 1);
407     if (r < 0)
408         return r;
409     r = read_block(diskbno, blk);
410     if (r < 0)
411         return r;
412     else
413         return 0;
414 }
```

图 5.6 file_get_block 函数

file_truncate_blocks 函数如图 5.7 所示。

```
598 static void
599 file_truncate_blocks(struct File *f, off_t newsize)
600 {
601     int r;
602     uint32_t bno, old_nblocks, new_nblocks;
603
604     // Hint: Use file_clear_block and/or free_block.
605     // LAB 5: Your code here.
606     old_nblocks = ROUNDUP(f->f_size, BLKSIZE) / BLKSIZE;
607     new_nblocks = ROUNDUP(newsize, BLKSIZE) / BLKSIZE;
608     for (bno = new_nblocks; bno < old_nblocks; bno++)
609         file_clear_block(f, bno);
610     if (new_nblocks <= NDIRECT && f->f_indirect != 0) {
611         free_block(f->f_indirect);
612         f->f_indirect = 0;
613     }
614 }
```

图 5.7 file_truncate_blocks 函数

file_flush 函数如图 5.8 所示。

```
633 void
634 file_flush(struct File *f)
635 {
636     // LAB 5: Your code here.
637
638     int r;
639     uint32_t nblock, bno, diskbno;
640     nblock = ROUNDUP(f->f_size, BLKSIZE) / BLKSIZE;
641     for (bno = 0; bno < nblock; bno++) {
642         r = file_map_block(f, bno, &diskbno, 0);
643         if (r < 0)
644             panic("file_flush: File map block failed.");
645         if (block_is_dirty(diskbno))
646             write_block(diskbno);
647     }
648 }
```

图 5.8 file_flush 函数

5.2 File System Access from Client Environments

5.2.1 Client-Side File Descriptors

Exercise 7. Implement `fd_alloc` and `fd_lookup`. `fd_alloc` finds an unused file descriptor number, and returns a pointer to the corresponding file descriptor table entry. Similarly, `fd_lookup` checks to make sure a given file descriptor number is currently active, and if so returns a pointer to the corresponding file descriptor table entry.

Exercise 8. Implement `open`. It must find an unused file descriptor using the `fd_alloc()` function we have provided, make an IPC request to the file server to open the file, and then map all the file's pages into the appropriate reserved region of the client's address space. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file server fail.

Use `gmake grade` to test your code.

Exercise 9. Implement `close`. It must first notify the file server of any pages it has modified and then make a request to the file server to close the file. When the file server is asked to close the file, it will write the new data to disk. (Be sure you understand why the file system cannot just rely on the `PTE_D` bits in its own mappings of the file's pages to determine whether or not those pages were modified.) Finally, the `close` function should unmap all mapped pages in the reserved file-mapping region for the previously-open file, to help catch bugs in which the application might try to access that region after the file is closed.

Use `gmake grade` to test your code.

首先，考虑文件 lib/fd.c 中的函数，fd_alloc 函数实现如图 5.9 所示。

```
50 int
51 fd_alloc(struct Fd **fd_store)
52 {
53     // LAB 5: Your code here.
54
55     int i;
56     struct Fd *tmp;
57     for (i = 0; i < MAXFD; i++) {
58         tmp = INDEX2FD(i);
59         if ((vpd[PDX(tmp)] & PTE_P) == 0 || (vpt[VPN(tmp)] & PTE_P) == 0) {
60             *fd_store = tmp;
61             return 0;
62         }
63     }
64     *fd_store = 0;
65     return -E_MAX_OPEN;
66 }
```

图 5.9 fd_alloc 函数

fd_lookup 的函数实现如图 5.10 所示。

```
74 int
75 fd_lookup(int fdnum, struct Fd **fd_store)
76 {
77     // LAB 5: Your code here.
78
79     if (fdnum >= 0 && fdnum < MAXFD) {
80         *fd_store = INDEX2FD(fdnum);
81         if ((vpd[PDX(*fd_store)] & PTE_P) == 0 || (vpt[VPN(*fd_store)] & PTE_P) == 0)
82             return -E_INVAL;
83         else
84             return 0;
85     }
86     else
87         return -E_INVAL;
88 }
```

图 5.10 fd_lookup 函数

然后，考虑 lib/file.c 中的函数。open 函数的函数实现如图 5.11 所示。

```
31 open(const char *path, int mode)
32 {
33     // LAB 5: Your code here.
34
35     struct Fd *fd;
36     int r;
37     r = fd_alloc(&fd);
38     if (r < 0)
39         return r;
40     r = fsipc_open(path, mode, fd);
41     if (r < 0)
42         return r;
43     r = fmap(fd, 0, fd->fd_file.file.f_size);
44     if (r < 0) {
45         fd_close(fd, 1);
46         return r;
47     }
48     else
49         return fd2num(fd);
50 }
```

图 5.11 open 函数

file_close 函数实现如图 5.12 所示。

```
54 static int
55 file_close(struct Fd *fd)
56 {
57     // Unmap any data mapped for the file,
58     // then tell the file server that we have closed the file
59     // (to free up its resources).
60
61     // LAB 5: Your code here.
62
63     int r;
64     r = funmap(fd, fd->fd_file.file.f_size, 0, 1);
65     if (r < 0)
66         return r;
67     r = fsipc_close(fd->fd_file.id);
68     if (r < 0)
69         return r;
70     else
71         return 0;
72 }
```

图 5.12 file_close 函数

5.2.2 Spawning Processes

Exercise 10. The skeleton for the `spawn` function is in `lib/spawn.c`. We will put off the implementation of argument passing until the next exercise. Fill it in so that it operates roughly as follows:

1. Create a new environment.
2. Allocate a stack at `USTACKTOP - BY2PG` using the provided `init_stack` function.
3. Load the program text, data, and bss at the appropriate addresses specified in the ELF executable. Don't forget to clear to zero any portions of these program segments that are *not* loaded from the executable file.
4. Initialize the child's register state using the new `sys_set_trapframe` system call.
5. Start it running from the entry point specified in the executable's ELF header.

Use `gmake grade` to test your code.

spawn 函数的实现相当复杂，但只要按照要求和注释的提示，实现应该问题不大。这里只给出 spawn 载入 ELF 文件的关键 for 循环，如图 5.13 所示。这里最需要注意的是对于可加载段 Text 段和不可加载段 Bss、Data 段，要分别考虑，处理方法是不一样的。对于 Bss 和 Data 段，也需要分别考虑。

```
110     for (; ph < eph; ph++) {
111         if (ph->p_type == ELF_PROG_LOAD) {
112             uintptr_t start = ROUNDDOWN(ph->p_offset, PGSIZE);
113             uintptr_t end;
114             uintptr_t va = ROUNDDOWN(ph->p_va, PGSIZE);
115             int i;
116             void *blk;
117             if ((ph->p_flags & ELF_PROG_FLAG_WRITE) == 0) {
118                 // Text
119                 end = ROUNDUP(ph->p_filesz + ph->p_offset, PGSIZE);
120                 for (i = start; i < end; i += PGSIZE) {
121                     r = read_map(fdnum, i, &blk);
122                     if (r < 0)
123                         return r;
124                     r = sys_page_map(0, blk, child, (void *)(va + i - start), PTE_U | PTE_P);
125                     if (r < 0)
126                         return r;
127                 }
128             }
129             else {
130                 // Bss and Date
131                 uintptr_t limit = ph->p_offset + ph->p_filesz;
132                 end = ROUNDUP(ph->p_memsz + ph->p_offset, PGSIZE);
133                 seek(fdnum, ph->p_offset);
134                 for (i = start; i < end; i += PGSIZE) {
135                     r = sys_page_alloc(0, UTEMP, PTE_U | PTE_W | PTE_P);
136                     if (r < 0)
137                         return r;
138                     memset(UTEMP, 0, PGSIZE);
139                     if (i < limit) {
140                         // Data
141                         r = read(fdnum, UTEMP, PGSIZE);
142                         if (r < 0)
143                             return r;
144                         if (i == ROUNDDOWN(limit, PGSIZE))
145                             memset(UTEMP + limit - i, 0, PGSIZE - (limit - i));
146                     }
147                     r = sys_page_map(0, UTEMP, child, (void *)(va + i - start), PTE_U | PTE_W | PTE_P);
148                     if (r < 0)
149                         return r;
150                     r = sys_page_unmap(0, UTEMP);
151                     if (r < 0)
152                         return r;
153                 }
154             }
155         }
```

图 5.13 spawn 函数的关键部分

除此之外，还需要在 kern/syscall.c 中添加新的函数调用，如图 5.14 所示。

```
143 static int
144 sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
145 {
146     // LAB 4: Your code here.
147     // Remember to check whether the user has supplied us with a good
148     // address!
149
150     struct Env *env;
151     int err;
152     err = envid2env(envid, &env, 1);
153     if (err == 0) {
154         user_mem_assert(env, (void *)tf, sizeof(struct Trapframe), PTE_U);
155         env->env_tf = *tf;
156         env->env_tf.tf_cs = GD_UT | 3;
157         env->env_tf.tf_eflags != FL_IF;
158         return 0;
159     }
160     else
161         return err;
162 }
```

图 5.14 sys_env_set_trapframe 函数

5.2.3 Spawn arguments

> **Exercise 11.** We have set up `spawn()` so that it calls a helper function in the same source file, `init_stack()`, to set up the new child environment's stack. Most of the code for `init_stack()` is done for you; it allocates a temporary page and maps it into the parent's address space at a fixed address (from `TMPPAGE` through `TMPPAGETOP-1`), then (after the point at which you need to insert code) re-maps that page into the child's address space ending at `USTACKTOP`. You just need to copy the argument array and argument strings into the stack page at its temporary mapping in the parent, as indicated by the comments in the code. Be sure to change the line that sets `*init_esp` in order to give the child environment the correct initial stack pointer. The child's initial stack pointer should point to its 'argc' argument, as shown in the figure above.
>
> Use `gmake grade` to test your code.

这个 Exercise 主要是修改 init_stack 函数，修改后的 init_stack 如图 5.15 所示。

```
181 static int
182 init_stack(envid_t child, const char **argv, uintptr_t *init_esp)
183 {
184     size_t string_size;
185     int argc, i, r;
186     char *string_store;
187     uintptr_t *argv_store;
188
189     string_size = 0;
190     for (argc = 0; argv[argc] != 0; argc++)
191         string_size += strlen(argv[argc]) + 1;
192
193     string_store = (char*) UTEMP + PGSIZE - string_size;
194     // argv is below that.  There's one argument pointer per argument, plus
195     // a null pointer.
196     argv_store = (uintptr_t*) (ROUNDDOWN(string_store, 4) - 4 * (argc + 1));
197
198     if ((void*) (argv_store - 2) < (void*) UTEMP)
199         return -E_NO_MEM;
200
201     // Allocate the single stack page at UTEMP.
202     if ((r = sys_page_alloc(0, (void*) UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
203         return r;
204
205     // LAB 5: Your code here.
206
207     for (i = 0; i < argc; i++) {
208         argv_store[i] = UTEMP2USTACK(string_store);
209         strcpy(string_store, argv[i]);
210         string_store += strlen(argv[i]) + 1;
211     }
212     argv_store[argc] = 0;
213     *(argv_store - 1) = UTEMP2USTACK(argv_store);
214     *(argv_store - 2) = argc;
215     *init_esp = UTEMP2USTACK(argv_store - 2);
216
217     // After completing the stack, map it into the child's address space
218     // and unmap it from ours!
219     if ((r = sys_page_map(0, UTEMP, child, (void*) (USTACKTOP - PGSIZE), PTE_P | PTE_U | PTE_W)) < 0)
220         goto error;
221     if ((r = sys_page_unmap(0, UTEMP)) < 0)
222         goto error;
223
224     return 0;
225
226 error:
227     sys_page_unmap(0, UTEMP);
228     return r;
229 }
```

图 5.15 init_stack 函数

至此，Lab 5 的所有 Exercise 都已完成。完成后， 执行“sh grade.sh”进行评分，结果如图 5.16 所示。显然，评分为满分，表示 Lab 5 已经通过。

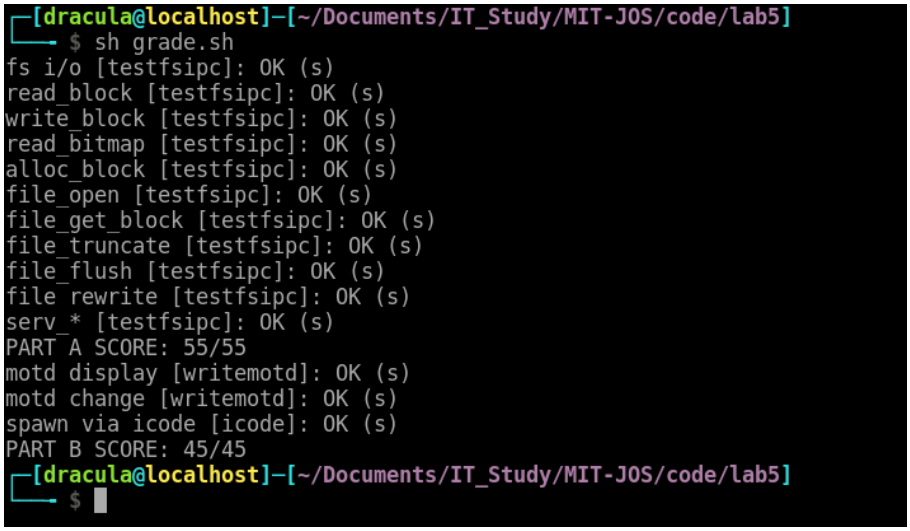

图 5.16 Lab 5 评分测试

6 Summary

6.1 Experiment Summary

本次操作系统的课程设计总共完成了 5 个 Lab，一直做到了 JOS 的文件系统的实现，可惜的是因为组原课设的压力，没有继续完成最后一个 Lab 来实现 JOS 的 Shell 实现完整的 JOS。虽说有点些许的遗憾，但总体来说，这次操作系统课设的成果还是让人满意的。

Lab 1 需要写的代码量比较少，但需要阅读的资料却有很多。Lab 1 中最复杂的部分实际上是 boot loader 的启动和执行，但这部分 JOS 已经实现好，并不需要动手自己写，只需看懂即可。当然，Lab 1 还涉及到 BIOS 的执行和作用、内核的加载和执行等等，概括来说，Lab 1 的核心就是如何启动一台 PC。除此之外，Lab 1 中还对 JOS 的输出函数（cprintf）作了介绍，并需要动手添加 8 进制数的打印；要求实现一条 backtrace 指令，这些都比较简单。

Lab 2 的重点则是内存管理，主要涉及三种地址——虚地址、线性地址和物理地址以及对应的转换机制——分段和分页。Lab 2 的任务就是实现一个段页式存储管理系统，但也就是这一部分，让我意识到实际的操作系统的段页式管理机制远比我们课本上学习的要复杂。Lab 2 的代码量不小，但如果能够理解前面提到的三种地址，并且掌握 JOS 的分段分页机制，Lab 2 就好实现一些。

Lab 3 实际上实现 JOS 进程创建和异常处理的功能，不过 JOS 中把“进程”这个概念用“environment”代替。JOS 的进程管理是分 Lab 3 和 Lab 4 两部分实现的，Lab 3 主要是实现了进程的创建、初始化等基本操作，具体的调度、通信等则是 Lab 4 实现。Lab 3 还基本实现了异常处理的功能，这一部分的关键是要理解 JOS 的异常处理机制，重点是到中断来临时，JOS 进行的操作和调用的函数是什么。

Lab 4 自我感觉是最难的一个 Lab。Lab 4 首先是要求实现轮转循环调度，这个倒不难，难点在后面实现 Copy on Write 以及 fork 函数。为了实现 Copy on Write 还需要实现用户级的缺页异常处理程序，为了实现这个功能又要了解用户异常栈的功能等等。同理，fork 函数的实现也想当复杂。为了做 Lab 4，真的是费了不少的心血。

Lab 5 实现文件系统本身难度并不大，但 Lab 5 的综合性很高，因为它要用到前

面 4 个 Lab 实现的所有功能，所以前面哪个 Lab 出了问题，都会造成文件系统出现这样或那样的 Bug，更麻烦的是这样的 Bug 十分隐蔽和不好定位，找这样的 Bug 占了 Lab 5 实现一半以上的时间。Lab 5 的实验报告因为时间的原因没有详细写。

6.2 Experiment Experience

JOS 是我上大学以来做过难度最大的一个课设：全英文的课程资料、庞大的代码阅读量和理解量、复杂的操作系统运行机制、让人抓狂的调试，哪一个对我而言都是不小的挑战。JOS 做到后面，真的是靠毅力在坚持。

但 JOS 也是让我印象很深刻的课设之一！通过做 JOS，我对操作系统尤其是 Linux 的理解更加深入了一步，而不仅仅局限于操作系统课程上所学到的东西。JOS 是学习操作系统的一次很好的实践，它将上个学期在课程中所学到的内容融会贯通，例如：段页式（分段和分页）内存管理机制，进程的创建、管理、调度和通信，文件系统的管理和操作，中断和异常处理等等。JOS 让我意识到，真正可用的操作系统用到的技术远比课本上介绍的要复杂，课本上介绍的只能说是一种设计思想和理念，真正的实现远比其要困难！

总而言之，这次操作系统课设收获颇丰！