華中科技大學

操作系统课程设计报告

学 生 姓 名: 胡思勖

学 号: U201514898

专 业: 计算机科学与技术

班 级: 计卓 1501

指 导 教 师: 张杰

2018-03-03

目录

第 1 章　Booting a PC

1.1　PC Bootstrap

这一部分主要介绍了 x86 语言以及 PC 的启动过程，并让我们熟悉了 QEMU 和 GDB 的调试方法。

1.1.1　Getting Started with x86 assembly

通过阅读 *PC Assembly Language Book*[1]来学习 x86 的语法。注意这本书值使用的 NASM 汇编，但是实验中使用的是 GNU 的 AT&T 语法。

> **Exercise 1**
> 了解和学习 x86 的语法。

Exercise 1 实验过程

由于已经上过 intel 汇编的理论课，因此这一部大部分为复习内容。通过阅读所给的参考文献，并结合以往的经验来看，较为重要的部分是熟悉汇编的语法，各个指令的作用以及能够掌握 C 语言与汇编的关系。

Exercise 1 实验过程

1.1.2　Simulating the x86

首先安装 qemu 虚拟机以及计算所使用的工具链，以及调试工具 GDB。此次实验使用的为 Arch Linux，且在实验过程中发现 gcc7.3.0 不能够正确的对 jos 源码进行编译，因此所使用的的 gcc 为从源码编译的 i386-jos-elf-gcc。QEMU 则是使用的 Arch Linux 官方源中的 qemu。

在安装完成后，进入目录并使用 make 进行编译。编译完成后使用 make qemu 命令启动 qemu。启动后的 QEMU 如图1.1所示。

可以发 jos 内核已经正常启动，并且能够执行 help 命令。执行 help 命令以后显示最初始的 jOS 由两个命令：help 以及 kerninfo。输入 kerninfo 命令以后命令行中打印出了部分有关内核的信息。

[1] https://pdos.csail.mit.edu/6.828/2017/readings/pcasm-book.pdf

```
Machine  View
SeaBIOS (version 1.11.0-20171110_100015-anatol)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92510+07EF2510 C980


Booting from Hard Disk...
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 1.1: 初次编译成功并启动的 qemu

1.1.3 The PC's Physical Address Space

从实验指导中可以知道，通常情况下一个 PC 的布局是如图1.2所示的。

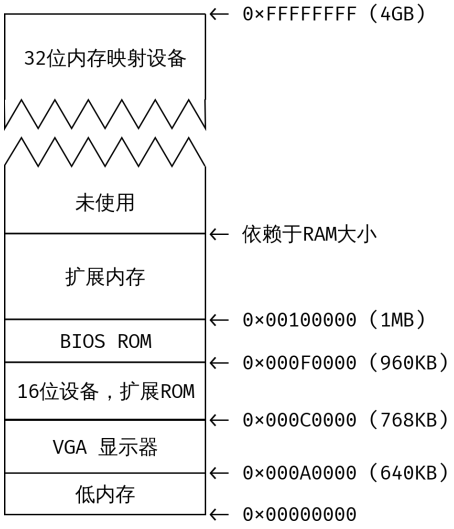

图 1.2: 内存布局

在早期的 16 位 8088 处理器只能够操作 1MB 物理内存，因此物理地址空间为 0x00000000 到 0x000FFFFF。其中 0~640KB 为 Low Memory，只能被 RAM 所使用。而从 0x000A0000 到 0x000FFFFF 的 384KB 内存保留做特殊用途，包括显示缓存以及其他固件。从 0x000F0000 到 0x000FFFFF 这 64KB 区域为 BIOS。现代 x86 处理器支持 4GB 的 RAM，因此 RAM 扩展到了 0xFFFFFFFF，但 jos 只使用开始的 256MB。

1.1.4 The ROM BIOS

在一个终端下执行 make qemu-gdb 指令，然后新建一个 terminal，执行 make gdb 指令，结果如图1.3所示（此处使用的是 cgb，便于调试，此后的调试也将使用 cgdb 而不是 gdb）。可以看到，gdb 已经开始调试并停止在 0xffff0 处。而这条指令就是 PC 启动后 BIOS 执行的第一条指令。

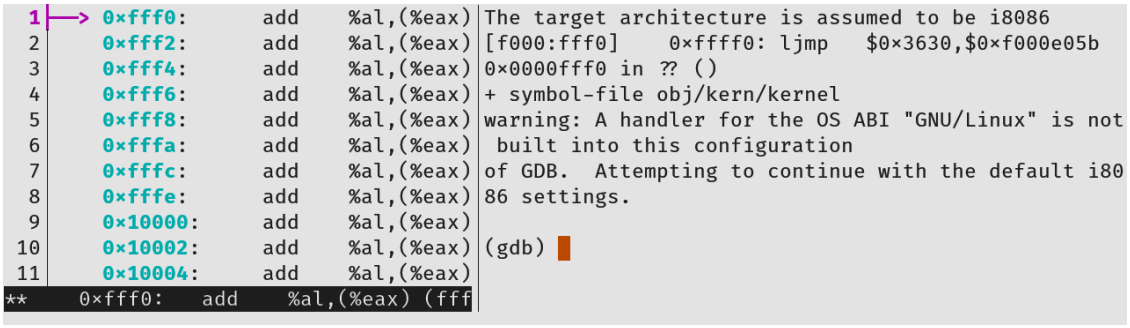

图 1.3: 初次进入调试界面

第一条指令表明：

- IBM PC 执行的起始物理地址为 0x000ffff0
- PC 的偏移方式为 CS = 0xf000，IP = 0xfff0
- 第一条指令执行的是 jmp 指令，跳转到段地址 CS = 0xf000，IP = 0xe05b

Exercise 2

使用 GDB 的 si 命令单步调试进入 ROM BIOS，然后猜测这些指令的作用。

Exercise 2 实验过程

在使用 make qemu-gdb 命令编译内核并启动 qemu，然后使用 make gdb 命令启动 gdb 后，使用 si 单步调试。连续执行 si 命令多步后，得到的前 30 条指令如下：

```
1  0xffff0:  ljmp   $0xf000, $0xe05b
2  0xfe05b:  cmpl   $0x0,    $cs:0x6ac8
3  0xfe062:  jne    0xfd2e1
4  0xfe066:  xor    %dx,     %dx
5  0xfe068:  mov    %dx,     %ss
6  0xfe06a:  mov    $0x7000, %esp
7  0xfe070:  mov    $0xf34d2,%edx
8  0xfe076:  jmp    0xfd15c
9  0xfd15c:  mov    %eax,    %ecx
10 0xfd15f:  cli
11 0xfd160:  cld
12 0xfd161:  mov    $0x8f,   %eax
13 0xfd167:  out    %al,     $0x70
14 0xfd169:  in     $0x71,   %al
15 0xfd16b:  in     $0x92,   %al
```

```
16  0xfd16d:  or     $0x2,    %al
17  0xfd16f:  out    %al,     $0x92
18  0xfd171:  lidtw  %cs:0x6ab8
19  0xfd177:  lgdtw  %cs:0x6a74
20  0xfd17d:  mov    %cr0,    %eax
21  0xfd180:  or     $0x1,    %eax
22  0xfd184:  mov    %eax,    %cr0
23  0xfd187:  ljmpl  $0x8,    $0xfd18f
24  0xfd18f:  mov    $0x10,   %eax
25  0xfd194:  mov    %eax,    %ds
26  0xfd196:  mov    %eax,    %es
27  0xfd198:  mov    %eax,    %ss
28  0xfd19a:  mov    %eax,    %fs
29  0xfd19c:  mov    %eax,    %gs
30  0xfd19e:  mov    %ecx,    %eax
```

其中:

- 第 1 条指令是一条跳转指令，跳转到 0xfe05b 处。
- 第 2 条和第 3 条指令共同构成了判断跳转，如果 $cs:0x6ac8 处的值不为 0 则跳转，而根据第 4 条指令的地址可以得知此处并没有发生跳转。
- 第 4 跳指令将%dx 清零。
- 第 5~8 条指令在设置完段寄存器、段指针以及 edx 寄存器后跳转到了 0xfd15c 处。然后第 10 跳指令关闭了中断。这应该是由于 boot 过程是比较关键的，因此屏蔽大部分的中断。
- 第 11 跳指令设置了方向标志位，表示后续操作的内存变化方向。
- 第 12~14 条指令涉及到 IO 操作。CPU 与外部设备通讯需要通过 IO 端口访问，而 x86CPU 使用 IO 端口独立编址的方式。并且规定端口操作需要通过 al 或者 ax 进行。通过查询端口对应的设备[2]我们知道了 0x70 和 0x71 是用于操作 CMOS 的端口。而这三条指令是用于关闭不可屏蔽中断 (Non-Maskable Interrupt) 的。
- 第 18 和 19 条指令用于将 0xf6ab8 处的数据读入到中断向量表寄存器 (IDTR) 中，并将 0xf6a74 的数据读入到全局描述符表格寄存器 (GDTR) 中，而后者是实现保护模式中较为重要的一部分。
- 第 20~21 用于将 CR0 寄存器的最低位置 1，进入保护模式，然而后面又从保护模式退出，继续在实模式下运行。
- 第 23~29 步用于重新加载段寄存器，在加载完 GDTR 寄存器后需要刷新所有的段寄存器的值[3] 。

Exercise 2 实验过程

[2] http://bochs.sourceforge.net/techspec/PORTS.LST

[3] https://en.wikibooks.org/wiki/X86_Assembly/Global_Descriptor_Table

1.2 The Bootloader

对于 PC 而言软盘和硬盘都被分为 512 字节的扇区。一个扇区是磁盘传输的最小粒度，也就是说读写都必须以扇区为单位执行。如果一个磁盘是可启动的，就把这个磁盘的第一个扇区叫做启动扇区，而 boot loader 就位于这一个扇区中。当 BIOS 找到一个可以启动的磁盘以后就会把这 512 字节的扇区加载到 0x7c00 到 0x7dff 中。

jOS 采用传统的硬盘启动机制，也就是说 boot loader 的大小小于 512 字节。bootloader 只包含两个文件：一个汇编文件 boot/boot.S 和一个 C 文件 boot/main.c，并且主要有如下两个功能：

- 将处理器由实模式转换为保护模式。
- 通过 x86 的特殊 I/O 访问指令将内核从硬盘读入内存。

Exercise 3

在 0x7c00 处设置断点，然后让程序运行到这个断点，然后使用 gdb 跟踪 boot.S 的每一条指令，并比较 obj/boot/boot.asm 以及 boot/boot.S。

追踪进入 bootmain() 中，然后追踪进入 readsect() 中，并找出每一条 readsect() 中 c 语句对应的汇编语句，然后找出把内核从磁盘读取到内存的 for 语句对应的汇编语句。找出 loop 运行完后那些语句会被执行，并在那里设置断点。继续运行到断点然后执行完接下来的语句。

Exercise 3 实验过程

首先使用 make qemu-gdb 命令启动 qemu，然后使用 make gdb 命令启动 gdb 并在 gdb 中使用 b *0x7c00 打上断点，然后使用 c 命令执行到此断点处。

从反汇编的代码 obj/boot/boot.asm 中可以看出，0x7c00 就是 boot loader 的起始地址。也就是说，实验要求我们从 boot loader 的开始处跟踪。

首先观察并分析 boot.S 以及 main.c 中的内容，分析每一个部分的作用，以便于后续的跟踪:

```
1  .globl start
2  start:
3    .code16                     # Assemble for 16-bit mode
4    cli                         # Disable interrupts
5    cld                         # String operations increment
6    # Set up the important data segment registers (DS, ES, SS).
7    xorw    %ax,%ax             # Segment number zero
8    movw    %ax,%ds             # -> Data Segment
9    movw    %ax,%es             # -> Extra Segment
10   movw    %ax,%ss             # -> Stack Segment
```

在 boot.S 的开始部分，首先关闭了在 BIOS 运行期间可能打开的中断，防止 boot loader 在运行期间被中断。cld 用于设置字符串处理时指针的移动方向。

从 xorw 开始的 4 句代码用于将段寄存器清零：首先将寄存器%ax 置为 0，然后将%ax 赋值给段寄存器%ds, %es, %ss。由于在 BIOS 阶段这些寄存器的值可能发生改变因此此处需要对其进行复位操作。

在上述准备代码完成后，执行从实模式到保护模式的代码：

```
1   # Enable A20:
2   #   For backwards compatibility with the earliest PCs, physical
3   #   address line 20 is tied low, so that addresses higher than
4   #   1MB wrap around to zero by default.  This code undoes this.
5 seta20.1:
6   inb     $0x64,%al               # Wait for not busy
7   testb   $0x2,%al
8   jnz     seta20.1
9
10  movb    $0xd1,%al               # 0xd1 -> port 0x64
11  outb    %al,$0x64
12
13 seta20.2:
14  inb     $0x64,%al               # Wait for not busy
15  testb   $0x2,%al
16  jnz     seta20.2
17
18  movb    $0xdf,%al               # 0xdf -> port 0x60
19  outb    %al,$0x60
```

这部分代码用于将 CPU 从实模式转换为保护模式：为了保留对于早期计算机的后向兼容性，第 20 根物理地址线保持为 0，因此所有大于 1MB 的地址全部会回卷，而这段代码解除了这一限制。

seta20.1 后面的 3 句指令不停地检测 0x64 端口的第 0x2 位，若其不为 0 则重复这一检测。这三句命令用于阻塞直到输入缓冲区为空。当缓冲区准备完成后，movb 和 outb 指令向 0x64 端口写入 0xd1。经过查询，0xd1 指令的作用是将下一次写入 0x60 端口的指令会被发送给键盘控制器 804x 的输入端口。

seta20.2 后面的三句与 seta20.1 后面的三句相同，再次等待缓冲区可以切入。在等待完成后的 movb 和 outb 指令向 0x60 端口写入数据 0xdf。0xdf 表示使能 A20 线，可以进入保护模式。

```
1   # Switch from real to protected mode, using a bootstrap GDT
2   # and segment translation that makes virtual addresses
3   # identical to their physical addresses, so that the
4   # effective memory map does not change during the switch.
5   lgdt    gdtdesc
6   movl    %cr0, %eax
7   orl     $CR0_PE_ON, %eax
8   movl    %eax, %cr0
9
10  # Jump to next instruction, but in 32-bit code segment.
11  # Switches processor into 32-bit mode.
12  ljmp    $PROT_MODE_CSEG, $protcseg
```

使能 A20 线后，首先把 gdtdesc 标识符送入 GDTR 中。GDTR 有 48 位长，其中高 32 位表示该表在内存中的起始地址，低 16 位表示该表的长度。这一指令就是将 GDT 表的起始地址以及长度送入 GDTR 中。gdtdsec 的标识符内容在 boot.S 尾部：

```
gdtdesc:
  .word   0x17                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

可以看出 gdt 表的大小为 0x17，表的起始地址为标号 gdt 所在的位置。

在加载完 GDT 表的信息到 GDTR 后，使用了三个指令修改给%cr0 寄存器的内容。从文件首可以找到：

```
.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag
```

也就是说，这几句将寄存器%cr0 的最低位置为 1，而%cr0 的最低位是保护模式的启动位，将 CR0 的这一位置为 1 则表示保护模式启动。

在正式启动保护模式后，执行了一个 ljmp 指令，将当前的运行模式切换为 32 为地址模式。接下来运行的就是 32 位的代码：

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

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain
```

在 32 位的代码开头，修改了 5 个段寄存器的值。与前述 BIOS 部分一样，当加载完 GDTR 后必须重新加载段寄存器的值，而加载 CS 的值必须通过 ljmp 加载（也就是将运行模式切换为 32 位地址模式的指令）。从而使 GDTR 生效。然后设置完栈指针的值后，跳转到 bootmain 中去。

在 main.c 中，bootmain 函数如下：

```
void
bootmain(void)
{
    struct Proghdr *ph, *eph;

    // read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC)
```

```
11          goto bad;
12
13      // load each program segment (ignores ph flags)
14      ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
15      eph = ph + ELFHDR->e_phnum;
16      for (; ph < eph; ph++)
17          // p_pa is the load address of this segment (as well
18          // as the physical address)
19          readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
20
21      // call the entry point from the ELF header
22      // note: does not return!
23      ((void (*)(void)) (ELFHDR->e_entry))();
24
25  bad:
26      outw(0x8A00, 0x8A00);
27      outw(0x8A00, 0x8E00);
28      while (1)
29          /* do nothing */;
30  }
```

从代码中可以看出，首先 bootmain 调用了 readseg 函数：

```
1  // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
2  // Might copy more than asked
3  void
4  readseg(uint32_t pa, uint32_t count, uint32_t offset)
```

而从 readseg 函数定义处的注释可以看出，这个函数的作用是：将从 offset 开始的 count 个字节读到 pa 所在的位置。因此在 bootmain 中，readseg((uint32_t) ELFHDR, SECTSIZE*8, 0); 这条语句的作用是将内核的第一页 ($SECTSIZE \times 8 = 4096$) 的内容读取到 ELFHDR 处，也就是说，把内核的 elf 头部放入内存开始部分。

然后，下一条语句验证这是否是一个合法的 elf 文件头部。如果不是则跳转到 bad 标签处。

然后，在 elf 头部找到程序的段信息。而 e_phoff 表示的是 ProgramHeaderTable 距离 elf 起始地址的偏移量。将这一偏移量加上 elf 的起始地址存入 ph 中，作为 ProgramHeaderTable 的起始地址，然后通过偏移量将 eph 作为其结束地址。

接下来的 for 循环就是加载所有的段进入内存中。p_offset 值得是这一段开头相对于 elf 文件开头的偏移量，ph_memsz 指的是这个段被装入内存后的大小，而 p_pa 指的则是装入内存的起始地址。

最后 ((void (*)(void)) (ELFHDR->e_entry))(); 语句执行 e_entry。e_entry 是执行的入口地址，也就是说，通过执行这个地址的方式把控制权从 bootloader 转移给内核。

接下来，开始使用 gdb 进行跟踪。首先设置断点，然后运行到断点处。gdb 反汇编代码如1.4所示，图中显示了 GDB 反汇编的起始 17 条指令

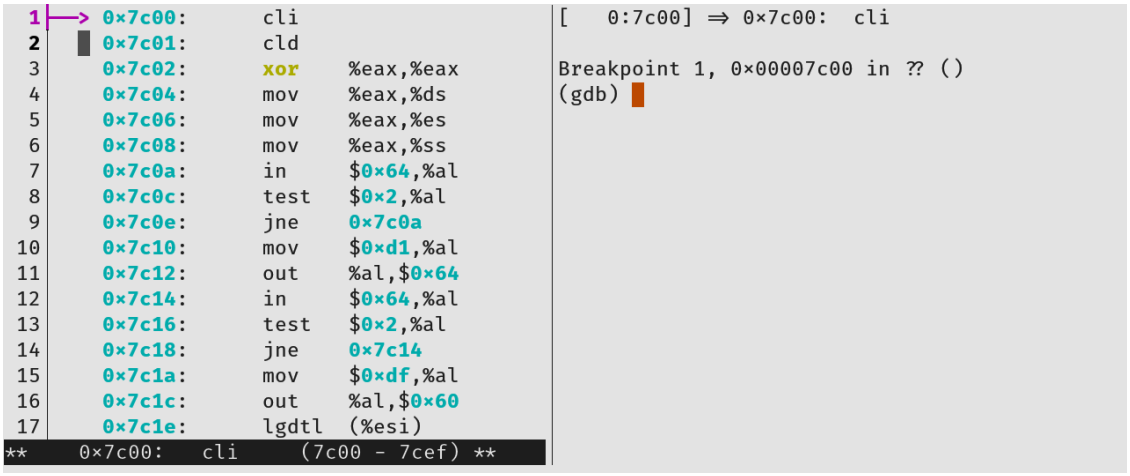

图 1.4: gdb 反汇编代码

下面为 boot/boot.S 的前 17 条指令

```
1  .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
2  .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
3  .set CR0_PE_ON,      0x1         # protected mode enable flag
4
5  .globl start
6  start:
7    .code16                     # Assemble for 16-bit mode
8    cli                         # Disable interrupts
9    cld                         # String operations increment
10
11   # Set up the important data segment registers (DS, ES, SS).
12   xorw    %ax,%ax             # Segment number zero
13   movw    %ax,%ds             # -> Data Segment
14   movw    %ax,%es             # -> Extra Segment
15   movw    %ax,%ss             # -> Stack Segment
16
17   # Enable A20:
18   #   For backwards compatibility with the earliest PCs, physical
19   #   address line 20 is tied low, so that addresses higher than
20   #   1MB wrap around to zero by default.  This code undoes this.
21 seta20.1:
22   inb     $0x64,%al               # Wait for not busy
23   testb   $0x2,%al
24   jnz     seta20.1
25
26   movb    $0xd1,%al               # 0xd1 -> port 0x64
```

然后是 obj/boot/boot.asm 中的前 10 条指令:

```
1  .globl start
```

```
 2  start:
 3    .code16                     # Assemble for 16-bit mode
 4    cli                         # Disable interrupts
 5      7c00:   fa                      cli
 6    cld                         # String operations increment
 7      7c01:   fc                      cld
 8
 9    # Set up the important data segment registers (DS, ES, SS).
10    xorw    %ax,%ax             # Segment number zero
11      7c02:   31 c0                   xor    %eax,%eax
12    movw    %ax,%ds             # -> Data Segment
13      7c04:   8e d8                   mov    %eax,%ds
14    movw    %ax,%es             # -> Extra Segment
15      7c06:   8e c0                   mov    %eax,%es
16    movw    %ax,%ss             # -> Stack Segment
17      7c08:   8e d0                   mov    %eax,%ss
18
19  00007c0a <seta20.1>:
20    # Enable A20:
21    #   For backwards compatibility with the earliest PCs, physical
22    #   address line 20 is tied low, so that addresses higher than
23    #   1MB wrap around to zero by default.  This code undoes this.
24  seta20.1:
25    inb     $0x64,%al               # Wait for not busy
26      7c0a:   e4 64                   in     $0x64,%al
27    testb   $0x2,%al
28      7c0c:   a8 02                   test   $0x2,%al
29    jnz     seta20.1
30      7c0e:   75 fa                   jne    7c0a <seta20.1>
31
32    movb    $0xd1,%al               # 0xd1 -> port 0x64
33      7c10:   b0 d1                   mov    $0xd1,%al
```

通过比较可以发现，源程序中如 movb，inb 等指令在 gdb 中实际执行的是 mov，in 指令，用于标识操作数长度的后缀没有了，但是指令的实际功能是相同的。在 boot/boot.S 中的标识在 gdb 中变为了实际的物理地址。而在 boot.asm 中则同时包含了这两项，即转换前使用标识标示地址以及编译后使用真实地址表示地址的。此外 boot/boot.S 中的注释是不被编译的。

继续执行，一直执行到 call bootmain 处。执行完后 gdb 反汇编如1.5所示。

bootmain 中，7d0b 到 7d0f 处的 4 条语句是 C 语言执行函数调用的时候必须要进行的一些任务，包括修改栈帧信息以及保护寄存器%esi，%ebx 的值。在进行了这些必要的任务后。7d10 到 7d12 处的指令是对于参数的压栈，以将参数传递给 regadseg 函数。压栈完成后使用 call 语句调用 readseg 函数。

Excercise 3 要求追踪到 readsect 函数中，因此继续向下追踪。执行到 readseg 函数时，反汇编如图1.6所示。

在 readseg 函数中，7cd2 到 7cdb 处的程序依然是例行执行的修改栈帧、保护寄存器、初始化局部

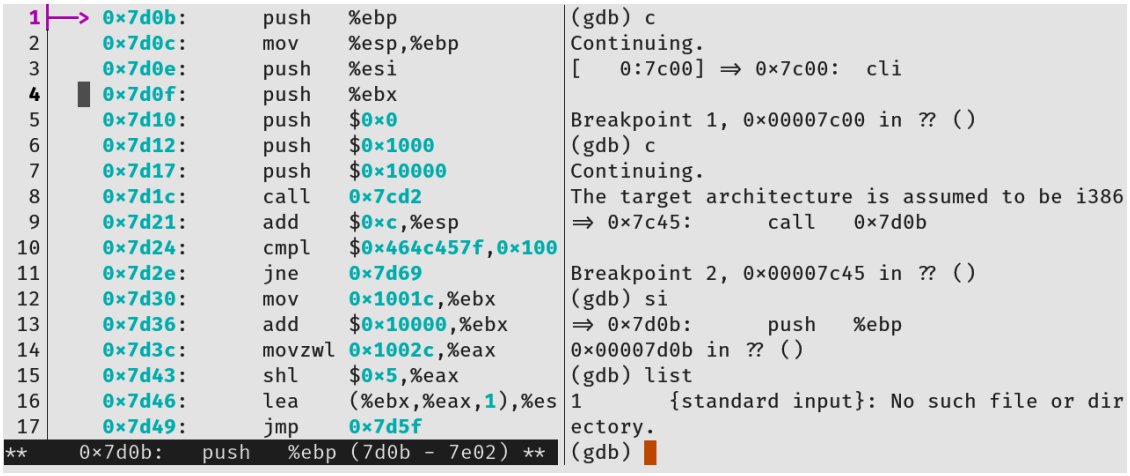

图 1.5: gdb 中 bootmain 的反汇编

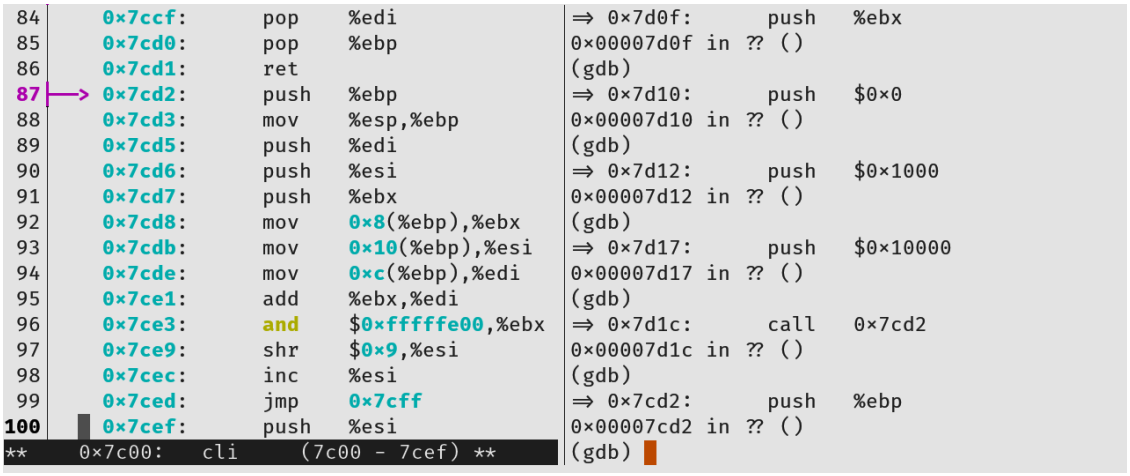

图 1.6: gdb 中 readseg 函数的反汇编

变量等。7cde 到 7cec 处的汇编语句用于计算 end_pa、pa、offset，为后面读取内核做准备。从这三句汇编语言可以看出一句 C 语言语句可能对应不止一句汇编语句。

在计算完起始地址，结束地址，偏移量后，进入 while 循环。在进入循环的第一条指令为 jmp 7cff，进行了一个绝对跳转。实际上，C 语言实现 while 循环一般是先进行跳转，将判断的语句置于循环体之后，也就是将 while 实现为 do-while 的形式。进行判断的语句为：

```
1 7cff: 39 fb cmp %edi,%ebx
2 7d01: 72 ec jb 7cef <readseg+0x1d>
```

在进行了判断后，跳转到 7cef，而 7cef 正是循环体的开始地址。在循环体中，一共执行了 3 条 C 语言语句：

```
1 readsect((uint8_t*) pa, offset);
2 pa += SECTSIZE;
3 offset++;
```

从这三条语句中可以看出，readseg 是通过调用 readsect 来逐个扇区将数据读入内存从而达到加载整个段的效果。下面使用 gdb 继续追踪进入 readsect 中。此时 gdb 的反汇编如1.7所示。

```
51 ─> 0x7c81:      push   %ebp              ⇒ 0x7cff:      cmp    %edi,%ebx
52    0x7c82:      mov    %esp,%ebp         0x00007cff in ?? ()
53    0x7c84:      push   %edi              (gdb)
54    0x7c85:      mov    0xc(%ebp),%edi    ⇒ 0x7d01:      jb     0x7cef
55    0x7c88:      call   0x7c6c            0x00007d01 in ?? ()
56    0x7c8d:      mov    $0x1f2,%edx       (gdb)
57    0x7c92:      mov    $0x1,%al          ⇒ 0x7cef:      push   %esi
58    0x7c94:      out    %al,(%dx)         0x00007cef in ?? ()
59    0x7c95:      mov    $0xf3,%dl         (gdb)
60    0x7c97:      mov    %edi,%eax         ⇒ 0x7cf0:      push   %ebx
61    0x7c99:      out    %al,(%dx)         0x00007cf0 in ?? ()
62    0x7c9a:      mov    %edi,%eax         (gdb)
63    0x7c9c:      shr    $0x8,%eax         ⇒ 0x7cf1:      call   0x7c81
64    0x7c9f:      mov    $0xf4,%dl         0x00007cf1 in ?? ()
65    0x7ca1:      out    %al,(%dx)         (gdb)
66    0x7ca2:      mov    %edi,%eax         ⇒ 0x7c81:      push   %ebp
67    0x7ca4:      shr    $0x10,%eax        0x00007c81 in ?? ()
**    0x7c00:   cli     (7c00 - 7cef) **  (gdb)
```

图 1.7: readsect 在 gdb 中的反汇编

从反汇编中可以看出，readsect 函数体执行的第一个语句为 waitdisk。而根据注释可以知道这一语句用于等待磁盘空闲。在磁盘空闲后，执行了一系列 outb 语句，这一系列 outb 语句用于向端口写值。第一个 outb 语句对应的汇编为：

```
1  7c8d:   ba f2 01 00 00          mov    $0x1f2,%edx
2  7c92:   b0 01                   mov    $0x1,%al
3  7c94:   ee                      out    %al,(%dx)
```

可以看出，C 语言中 outb 的实现就是使用汇编的 out 语句实现的端口 IO。从所有 outb 语句对应的 7c8d 到 7cb9 之间的汇编语句可以看出：outb 语句中第一个参数是端口号，第二个参数是送入端口的值。然后经过查询得知，这一函数首先向端口 0x1f2 送入值 1，这表示取出一个扇区；然后将要取出的扇区的 32 位编号分为 4 段，分别送入 0x1f3 到 0x1f6，最后向 0x1f7 端口送入 0x20 指令，表示要读取这个扇区。

在上述的一系列动作完成后，调用 waitdisk() 阻塞并等待磁盘读取完成。读取完成后继续执行下一条语句，即 insl 语句。insl 函数对应的汇编指令如下。

```
1  7cbf:   8b 7d 08                mov    0x8(%ebp),%edi
2  7cc2:   b9 80 00 00 00          mov    $0x80,%ecx
3  7cc7:   ba f0 01 00 00          mov    $0x1f0,%edx
4  7ccc:   fc                      cld
5  7ccd:   f2 6d                   repnz insl (%dx),%es:(%edi)
```

可以看出，insl 函数有 3 个参数，而在第一条指令中，将 insl 的第一个参数送入 edi。第一个参数是 dst，也就是目的地址。0x7cc2 到 0x7cc7 处的两条指令指令将寄存器%ecx 和%edx 分别赋值为 0x80 和 0x1f0。在执行 cld，即清除方向标识指令后，执行了一个 repnz 指令，repnz 指令重复执行紧随其后的指令直到寄存器%ecx 的值为 0 且 ZF 标志位为 1。在这里，被重复的指令是 insl (%dx), %es:(%edi)。在这个指令中，%dx 为端口号，此处为 0x1f0，%edi 为起始地址 pa，此时为 0x10000。单步执行这一指令一次，前后 0x10000 处的数据如1.8所示。

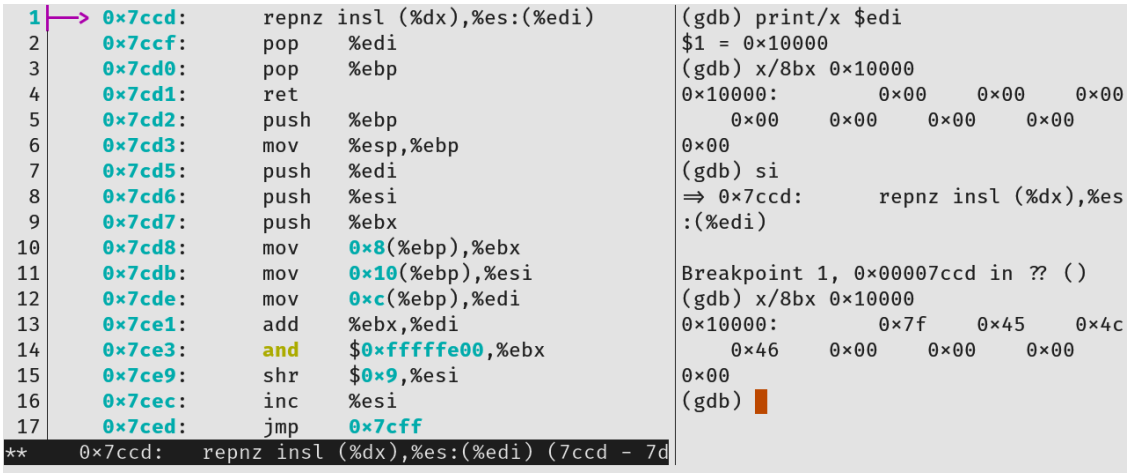

图 1.8: 执行一条指令前后 0x10000 处的数据

可以看到，一条 insl 指令读取 4 字节的数据。因此读取一个扇区（512）字节的读取需要重复 128 次 insl 指令，执行到下一条指令时%edi 的值也验证了这一点。

当读取完成后，执行如下三条指令，其作用是还原之前保护的寄存器的值并返回到调用者处。

```
1 7ccf:   5f                    pop    %edi
2 7cd0:   5d                    pop    %ebp
3 7cd1:   c3                    ret
```

回到 readseg 函数后，执行调用 readsect 完成后接下来的指令：

```
1     pa += SECTSIZE;
2 7cf6:   81 c3 00 02 00 00     add    $0x200,%ebx
3     offset++;
4 7cfc:   46                    inc    %esi
5 7cfd:   58                    pop    %eax
6 7cfe:   5a                    pop    %edx
7 offset = (offset / SECTSIZE) + 1;
8
9 // If this is too slow, we could read lots of sectors at a time.
10 // We'd write more to memory than asked, but it doesn't matter --
11 // we load in increasing order.
12 while (pa < end_pa) {
13 7cff:   39 fb                 cmp    %edi,%ebx
14 7d01:   72 ec                 jb     7cef <readseg+0x1d>
```

这些代码的作用在分析部分已经提到，其中 0x7cf6 和 0x7cfc 处的代码就是 pa+=SECTSIZE 以及 offset++ 两条 C 语言代码的直接对应，紧随其后的两条 pop 则是释放先前传入 readsect 的参数。最后的 cmp 以及 jb 则实现了 while 循环。

在 while 循环结束后的第一条语句打上断点，此处 (0x7d03~0x7d0a) 处的 5 条语句就是例行的恢复保护的寄存器以及返回到调用者处。之后，追踪返回到 bootmain 中。通过 boot.asm 可以判断出，在判断完 ELF 头部是否合法后，接下来的 for 语句的起始和结束位置分别为 0x7d49 以及 0x7d5f~0x7d61

处。C 语言 for 循环对应的汇编实现也是实现为 do-while 形式，与 while 类似，首先进行一次跳转。跳转到循环尾部再进行判断，如果判断成立则跳转到循环体开头处执行循环体，如果不是则直接执行下一语句，跳出循环。

```
for (; ph < eph; ph++)
7d49:   eb 14                   jmp    7d5f <bootmain+0x54>
    // p_pa is the load address of this segment (as well
    // as the physical address)
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
7d4b:   ff 73 04                pushl  0x4(%ebx)
7d4e:   ff 73 14                pushl  0x14(%ebx)
7d51:   ff 73 0c                pushl  0xc(%ebx)
7d54:   e8 79 ff ff ff          call   7cd2 <readseg>
    goto bad;

// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
7d59:   83 c3 20                add    $0x20,%ebx
7d5c:   83 c4 0c                add    $0xc,%esp
7d5f:   39 f3                   cmp    %esi,%ebx
7d61:   72 e8                   jb     7d4b <bootmain+0x40>
```

0x7d4b~0x7d59 执行的就是 for 循环的循环体，也仅对应一条 C 语言语句，即 readseg(ph->p_pa, ph->p_memsz, ph->p_offset); 前面已经分析过 readseg 函数的执行过程，而这一语句，就是 elfheader 中所指向的信息加载到 ph->p_pa 处。

在循环执行完成后，只执行了一条语句：

```
((void (*)(void)) (ELFHDR->e_entry))();
7d63:   ff 15 18 00 01 00       call   *0x10018
```

从 C 语言的语法分析，就是将 ELFHDR->e_entry 强制转化为 void (*)(void) 类型的函数指针然后执行这一函数指针，也就是说，将 CPU 的控制权从 bootloader 转移给内核。在 0x7d63 处打上断点，进入内核后执行的第一条语句如1.9所示。也就是 0x10000c 处的 movw $1234, 0x472。

Exercise 3 实验过程

至此，对于整个 bootloader 的分析与追踪已经完成。对于实验指导中提出的 4 个问题，其答案在 Exercise 3 的分析与追踪的过程中已经给出：

1. 在什么时候处理器开始工作于 32 位模式? 是什么让 CPU 从 16 位模式切换到 32 位模式?

   - 执行完 0x7c2d 处的 ljmp $0xb866, $0x87c32（对应 boot.S 中 ljmp $PROT_MODE_CSEG, $protcseg）语句后，处理器开始工作于 32 位模式。表面上，ljmp 语句将 CPU 切换为 32 位模式，实际上是因为此时 CPU 工作于保护模式下。

2. boot loader 中执行的最后一条语句是什么? 内核加载完成后执行的第一条语句又是什么?

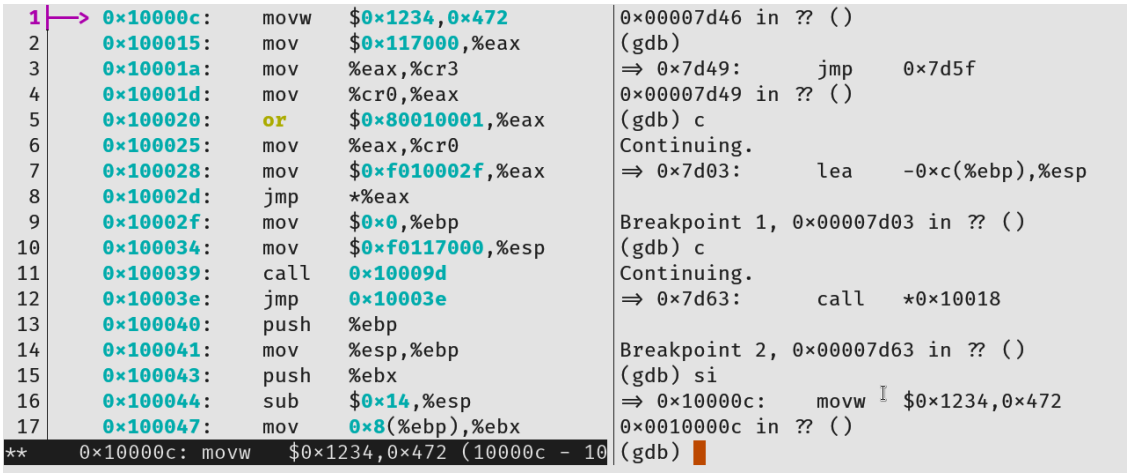

图 1.9: 进入内核后执行的第一条语句

- boot loader 执行的最后一条语句是 0x7d63 处的 call *0x10018，对应 main.c 中的 ((void (*)(void)) (ELFHDR->e_entry))(); 这条语句执行完后，执行的第一条指令，也就是加载完内核后执行的第一条指令是 movw $0x1234, 0x472。

3. 内核的第一条指令在哪里?

   - 内核的第一条指令在 0x1000c 处，对应的源码位于 kern/entry.S 中。

4. boot loader 是如何知道他要加载多少扇区才能吧整个内核加载进入内存的? 它是从哪里找到这些信息的?

   - 这些信息位于操作系统的 Program Header Table 中，这个表存放在 ELF Header 中，通过读取这些信息就知道要读取多少扇区才能把整个内核送入内存。

1.2.1 Loading the Kernel

在即进一步讨论 boot loader 的 C 语言部分之前先回顾一些有关 C 语言的基础知识。

Exercise 4

阅读并学习关于 C 语言指针的知识（最好的参考书籍是 K&R）。

阅读 K&R 5.1 章到 5.5 章的内容，下载 pointer.c[a]，然后编译运行它。确保你理解了所有打印出来的值是从哪里来的，尤其是第 1 行和第 6 行的指针以及第二行到第 4 行的值，以及为什么第 5 行的值看题来像是程序崩溃了。

[a] https://pdos.csail.mit.edu/6.828/2017/labs/lab1/pointers.c

Exercise 4 实验过程

首先，下载，编译并运行 poninters.c（编译为 32 位可执行文件），其过程如1.10所示。

观察源程序。首先程序声明了 1 个数组 int a[4]、2 个指针 int *b, int *c 以及一个整形值 i。然后执行了第一条语句:

```
~/Playground ls                                Sun Feb 18 20:59:09 2018
pointers.c
~/Playground gcc -m32 pointers.c -o test
~/Playground ls                                Sun Feb 18 20:59:24 2018
pointers.c  test*
~/Playground ./test                            Sun Feb 18 20:59:25 2018
1: a = 0×ff9d962c, b = 0×5663e160, c = 0×f7f6d1e8
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0×ff9d962c, b = 0×ff9d9630, c = 0×ff9d962d
~/Playground                                   Sun Feb 18 21:00:56 2018
```

图 1.10: 编译 pointers.c 以及运行的过程

```
1  printf("1: a = %p, b = %p, c = %p\n", a, b, c);
```

显然，输出的 a,b,c 分别为 a 的首地址 0xff9d962c，molloc 为 b 分配的地址 0x5663e160 以及未初始化的指针 c 的地址 0xf7f6d1e8。

在此之后直到打印第二行之前，执行了如下的语句：

```
1  c = a;
2  for (i = 0; i < 4; i++)
3      a[i] = 100 + i;
4  c[0] = 200;
```

首先，让指针 c 指向 a 的首地址，然后使用一个 for 语句令数组 a 的四个元素的值变为 100, 101, 102, 103, 最后，将 c[0] 置为 200。由于 c 只想的即 a 的首地址，因此修改的就是 a[0] 的值，因此 a[0] = 200。a[1], a[2], a[3] 的值维持不变。

然后直到打印第三行之前，执行了如下语句：

```
1  c[1] = 300;
2  *(c + 2) = 301;
3  3[c] = 302;
```

这是 c 语言访问数组的 3 中不同的方法。由于在前一步中已经将 c 指向数组 a 的首地址，因此此处修改 c[1], *(c+2), 3[c] 的值实际上就是在修改 a[1], a[2], a[3] 的值。修改完成后 a[1]=300, a[2]=301, a[3]=302。

在打印第 4 行之前执行的语句为：

```
1  c = c + 1;
2  *c = 400;
```

这两条语句将 c 指向了 a[1] 所在的地址并修改了 c 所在地址处的值。即将 a[1] 修改为了 400，a[0], a[2], a[3] 的值没有变。

在打印第 5 行之前执行的语句为：

```
1  c = (int *) ((char *) c + 1);
2  *c = 500;
```

第一句：首先将 int 型指针 c 强制转换为 char 型指针，然后将此指针值加上 1，然后再强制转换回 int 型指针。从图1.10可以推断出，在执行这一语句前 a[1] 的地址为 0xff9d962c+0x4=0xff9d9630，强制转换为 char 型指针后加上 1，得到的值为 0xff9d9631（加上了一个 char 的大小），然后再强制转换回 int 型指针。又由于 x86CPU 采用的是小端存储，因此此时修改 *c 的值相当于修改了 a[1] 的高 3 个字节以及 a[2] 的最低字节，分别修改为 500 的低 3 个字节以及 500 的最高字节。因此修改后 a[1]=128144, a[2]=256, a[0] 和 a[3] 不变，修改前后各个字节的值如表1.1所示。

表 1.1: a[1] 以及 a[2] 各字节变换

数组	a[1]				a[2]			
地址 (偏移量 0xff9d9600)	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
值（更改前）	90	01	00	00	2d	01	00	00
值（更改后）	90	**f4**	**01**	**00**	**00**	01	00	00

在打印最后一行前执行的操作为：

```
1  b = (int *) a + 1;
2  c = (int *) ((char *) a + 1);
```

这两条语句将指针 b 指向 a[1] 的地址，将指针 c 再按照打印第 5 行的方法加设置为 a[0] 的地址上 1 个 char 的长度。于是指针 b 的值变为 0xff9d9630，指针 c 的值变为 0xff9d962d。

Exercise 4 实验过程

为了能够理解 boot/main.c，首先需要知道 ELF 文件的结构。ELF 文件由 3 部分组成：带有加载信息的文件头，程序段表以及程序段。每一个段都是一块连续的代码或者数据。它们在运行时首先要被加载到内存中，而 boot loader 的任务就是要把它们加载到内存中。对于 jOS 而言，我们对 3 个段非常感兴趣：

- .text 段：存放程序的可执行代码
- .rodata 段，存放所有的只读数据，如字符串常量
- .data 段，存放所有被初始化后的数据段，比如有初值的全局变量。

通过 objdump 命令可以得到有关这些段的信息。运行 objdump -h obj/kern/kernel 命令后得到如下结果：

```
       obj/kern/kernel:     file format elf32-i386

   Sections:
   Idx Name          Size      VMA       LMA       File off  Algn
     0 .text         0000178e  f0100000  00100000  00001000  2**2
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000704  f01017a0  001017a0  000027a0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000044e9  f0101ea4  00101ea4  00002ea4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00008c34  f010638d  0010638d  0000738d  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f010f000  0010f000  00010000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0119300  00119300  0001a300  2**5
                  ALLOC
  6 .comment      00000011  00000000  00000000  0001a300  2**0
                  CONTENTS, READONLY
```

可以看出，其中不仅仅包含.text, .rodata 以及.data 段，还包含一些其他的段信息。在这些信息中对于每一个段都有一个 VMA 以及一个 LMA，VMA 就是这个段的逻辑地址，而 LMA 则为这个段被加载到内存后的物理地址。

通过 objdump -x obj/kern/kernl 命令，我们可以看到 ELF 中哪些部分将被加载到内存，以及被加载到内存中的哪个地址，其输出如下：

```
obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
    filesz 0x0000efc1 memsz 0x0000efc1 flags r-x
    LOAD off    0x00010000 vaddr 0xf010f000 paddr 0x0010f000 align 2**12
    filesz 0x0000a300 memsz 0x0000a944 flags rw-
...
```

Program Header 中列出的就是所有被加载到内存中的段的信息，也就是 Program Headers Table 中的表项，所有而需要被加载到内存的段都被标记为 LOAD。BIOS 会把 boot sector 加载到 0x7c00 处，这是 boot sector 的加载地址，也是其链接地址。我们可以通过 boot/Makefrag 中的-Ttext 修改链接地址。

Exercise 5

再次追中进入 boot loader 然后尝试辨认出哪些语句会在 boot loader 的链接地址变化后“坏掉”。在 boot/Makefrag 中修改链接地址为一个错误的地址，然后运行 make clean。用 make 重新编译整个 lab，并追踪进 boot loader 看发生了什么。最后别忘了把链接地址改回来并运行 make clean。

Exercise 5 实验过程

首先，运行 make clean 将之前的编译结果清除掉，然后修改 boot/Makefrag 中的链接地址。Makefrag 中，文件 $(OBJDIR)/boot/boot 的编译命令中-Ttext 0x7C00 即为链接地址。将其修改为 0x1000 后重新运行 make 进行编译。编译完成后 obj/boot/boot.asm 中程序的前数行如下：

```
1  .globl start
2  start:
3    .code16                     # Assemble for 16-bit mode
4    cli                         # Disable interrupts
5      1000:   fa                      cli
6    cld                         # String operations increment
7      1001:   fc                      cld
```

可以看出，程序的虚拟地址已经从的 0x1000 而不是 0x7c00 开始了。BIOS 仍然会把 boot loader 加载到 0x7c00 处，因此在 0x7c00 处打上打断点并调试。执行到断点后，gdb 显示如图1.11所示。可以看到，前几句程序依然是正常执行。

```
1 ─> 0x7c00:      cli
2    0x7c01:      cld
3    0x7c02:      xor    %eax,%eax
4    0x7c04:      mov    %eax,%ds
5    0x7c06:      mov    %eax,%es
6    0x7c08:      mov    %eax,%ss
7    0x7c0a:      in     $0x64,%al
8    0x7c0c:      test   $0x2,%al
9    0x7c0e:      jne    0x7c0a
10   0x7c10:      mov    $0xd1,%al
11   0x7c12:      out    %al,$0x64
12   0x7c14:      in     $0x64,%al
13   0x7c16:      test   $0x2,%al
14   0x7c18:      jne    0x7c14
15   0x7c1a:      mov    $0xdf,%al
16   0x7c1c:      out    %al,$0x60
17   0x7c1e:      lgdtl  (%esi)
**    0x7c00:   cli     (7c00 - 7cef) **
```

```
6
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf00
0e05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
warning: A handler for the OS ABI "GNU/Linux"
 is not built into this configuration
of GDB.  Attempting to continue with the defa
ult i8086 settings.

(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb)
```

图 1.11: 重新编译后执行到断点时的 gdb 显示

接下来单步执行，在执行到 ljmp 指令时，本应该直接执行 ljmp 的下一条指令地址，但是这里由于链接地址的错误直接跳转到了 0xe05b，如1.12所示，导致了错误，程序不能够继续运行。

Exercise 5 实验过程

对于内核而言，其加载地址和链接地址是不同的。这和 boot loader 不一样。bootloader 将内核加载在低地址处，但是内核运行在高地址处。通过 objdump -f obj/kern/kernel 可以看到 ELF 头部中的 e_entry 字段。这个字段存放的就是可执行程序入口的链接地址。这一命令输出如下：

```
obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

```
 1 ┌─> 0×e05b:      add    %al,(%eax) [   0:7c1e] ⇒ 0×7c1e:  lgdtl  (%esi)
 2 │   0×e05d:      add    %al,(%eax) 0×00007c1e in ?? ()
 3 │   0×e05f:      add    %al,(%eax) (gdb)
 4 │   0×e061:      add    %al,(%eax) [   0:7c23] ⇒ 0×7c23:  mov    %cr0,%eax
 5 │   0×e063:      add    %al,(%eax) 0×00007c23 in ?? ()
 6 │   0×e065:      add    %al,(%eax) (gdb)
 7 │   0×e067:      add    %al,(%eax) [   0:7c26] ⇒ 0×7c26:  or     $0×1,%ax
 8 │   0×e069:      add    %al,(%eax) 0×00007c26 in ?? ()
 9 │   0×e06b:      add    %al,(%eax) (gdb)
10 │   0×e06d:      add    %al,(%eax) [   0:7c2a] ⇒ 0×7c2a:  mov    %eax,%cr0
11 │   0×e06f:      add    %al,(%eax) 0×00007c2a in ?? ()
12 │   0×e071:      add    %al,(%eax) (gdb)
13 │   0×e073:      add    %al,(%eax) [   0:7c2d] ⇒ 0×7c2d:  ljmp   $0×b866,$0×81032
14 │   0×e075:      add    %al,(%eax) 0×00007c2d in ?? ()
15 │   0×e077:      add    %al,(%eax) (gdb)
16 │   0×e079:      add    %al,(%eax) [f000:e05b]    0×fe05b: cmpw   $0×ff88,%cs:(%esi)
17 │   0×e07b:      add    %al,(%eax) 0×0000e05b in ?? ()
** └─  0×e05b:   add    %al,(%eax) (e05 (gdb)
```

图 1.12: ljmp 指令跳转后 gdb 显示

Exercise 6

使用 GDB 的 x 命令可以检查内存中的内容。使用 x/Nx ADDR 可以查看 ADDR 处的 N 个字的内容。字的大小是没有一个标准的，但是在 GDB 中，一个字为 2 个字节。重置机器（退出 QEMU/GDB 并重启，然后分别在 BIOS 进入 bootloader 以及 bootloader 进入 kernel 时检查 0x00100000 处的 8 个字。为什么这些内容是不同的？第二个断点处的内容是什么？

Exercise 6 实验过程

首先重新编译并启动 qemu/gdb，然后在 0x7c00 处打上断点。运行到断点处，使用 x/8x 0x00100000 命令检查 0x00100000 处的 8 个字。其结果如图1.13所示。

```
 1 ┌─> 0×7c00:      cli              Continuing.
 2 │   0×7c01:      cld              [   0:7c00] ⇒ 0×7c00:  cli
 3 │   0×7c02:      xor    %eax,%eax
 4 │   0×7c04:      mov    %eax,%ds  Breakpoint 1, 0×00007c00 in ?? ()
 5 │   0×7c06:      mov    %eax,%es  (gdb) x/8x 0×00100000
 6 │   0×7c08:      mov    %eax,%ss  0×100000:       0×00000000      0×00000000
 7 │   0×7c0a:      in     $0×64,%al 0×00000000      0×00000000
 8 │   0×7c0c:      test   $0×2,%al  0×100010:       0×00000000      0×00000000
 9 │   0×7c0e:      jne    0×7c0a    0×00000000      0×00000000
** └─  0×7c00:   cli      (7c00 - 7cef) (gdb)
```

图 1.13: 从 BIOS 进入 boot loader 后 0x00100000 处的内容

Exercise 3 告诉我们，从 boot loader 进入内核后执行的第一条指令在 0x10000c 处，因此在 0x10000c 处打上断点。运行到这一断点并重新使用 x/8x 0x00100000 检查，其结果如图1.14所示。

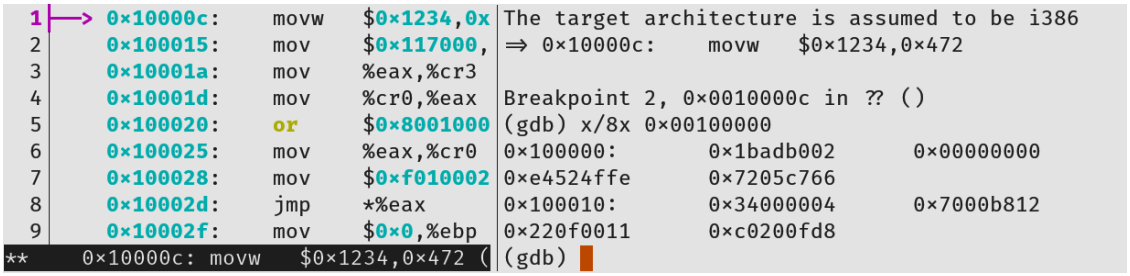

图 1.14: 从 boot loader 进入内核后 0x00100000 处的内容

观察 obj/kern/kernel.asm 的前几条指令，其指令内容如下：

```
1  .globl entry
2  entry:
3          movw    $0x1234,0x472           # warm boot
4  f0100000:   02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
5  f0100006:   00 00                   add    %al,(%eax)
6  f0100008:   fe 4f 52                decb   0x52(%edi)
7  f010000b:   e4 66                   in     $0x66,%al
8
9  f010000c <entry>:
10 f010000c:   66 c7 05 72 04 00 00    movw   $0x1234,0x472
11 f0100013:   34 12
```

可以看出，gdb 中 0x00100000 处的内容与此处的内容一致。此时可以回答 Exercise 6 提出的两个问题：此处内容不同是因为 boot loader 将内核加载到了 0x00100000 处，这 8 个字的内容就是内核的前若干条指令。

Exercise 6 实验过程

1.3 The Kernel

1.3.1 Using virtual memory to work around position dependence

在使用 boot loader 时，链接地址和加载地址是一样的，也就是说虚拟地址和物理地址是一样的。但是进入到内核程序后这两种地址就不再相同了。内核的虚拟地址会被链接到一个非常高的地址空间，对于 jOS 而言是 0xf0100000。剩下的较低部分的空间给其他的用户程序使用。

但是由于许多机器没有 0xf0100000 的物理内存，因此在运行的时候需要把虚拟地址映射到物理地址 0x00100000 处运行。此时加载的物理地址仅仅高于 BIOS ROM。此时 PC 仅最少需要 1MB 物理内存。

在这个实验中，采用分页管理的方法来实现映射，但是不是使用的通常的分页管理器，而是自己写了一个 lab/kern/entrygdir.c 进行映射。因此其功能很简单，只能够把 0x00000000~0xf0400000 以及 0x00000000~0x00400000 映射到 0x0000000~0x0-0400000 的范围内。不再这两个虚拟地址范围内的地址都会引起硬件异常。

Exercise 7

使用 QEMU 和 GDB 跟踪进入 JOS 的内核，并停在 movl %eax, %cr0 处，检查 0x00100000 处的内容以及 0xf0100000 处的内容。现在用 stepi 单步执行这个命令然后再检查 0x00100000 处的内容和 0xf0100000 处的内容，并确保你弄懂了刚才发生了什么。

如果不能正常建立映射，在新映射建立后最先不能够正常工作的指令是哪一条？注释掉 kern/entry.S 中的 movl %eax, %cr0 指令，然后看看你是否是正确的。

Exercise 7 实验过程

启动 qemu 和 gdb，由于 Exercise 3 已知程序的入口是 0x10000c，因此首先在 0x10000c 处打上断点并运行到 0x10000c 处，然后使用 x/8x 检查 0x00100000 和 0xf100000 处的内容，其内容如图1.15所示。

```
  1     0x10000c:    movw   $0x1234,0x472   (gdb) x/8x 0x00100000
  2     0x100015:    mov    $0x117000,%eax  0x100000:       0x1badb002      0x00000000
  3     0x10001a:    mov    %eax,%cr3       0xe4524ffe      0x7205c766
  4     0x10001d:    mov    %cr0,%eax       0x100010:       0x34000004      0x7000b812
  5     0x100020:    or     $0x80010001,%e  0x220f0011      0xc0200fd8
  6 ---> 0x100025:   mov    %eax,%cr0       (gdb) x/8x 0xf0100000
  7     0x100028:    mov    $0xf010002f,%e  0xf0100000 <_start+4026531828>: 0x00000000
  8     0x10002d:    jmp    *%eax           0x00000000      0x00000000      0x00000000
  9     0x10002f:    mov    $0x0,%ebp       0xf0100010 <entry+4>:   0x00000000      0x000000
 10     0x100034:    mov    $0xf0117000,%e  00      0x00000000      0x00000000
**    0x10000c: movw   $0x1234,0x472 (1000  (gdb)
```

图 1.15: 执行 movl %eax, %cr0 前的两处内存中的内容

由 Exercise 6 已知，0x00100000 处的内容就是内核的前几条指令，而从图1.15中可知，0xf0100000 处的内容全为 0。继续执行一步 stepi 以后，在次运行两个 x/8x 指令，这时显示的内容如1.16所示。

```
  2     0x100015:    mov    $0x117000,%eax  (gdb) x/8x 0x00100000
  3     0x10001a:    mov    %eax,%cr3       0x100000:       0x1badb002      0x00000000
  4     0x10001d:    mov    %cr0,%eax       0xe4524ffe      0x7205c766
  5     0x100020:    or     $0x80010001,%e  0x100010:       0x34000004      0x7000b812
  6     0x100025:    mov    %eax,%cr0       0x220f0011      0xc0200fd8
  7 ---> 0x100028:   mov    $0xf010002f,%e  (gdb) x/8x 0xf0100000
  8     0x10002d:    jmp    *%eax           0xf0100000 <_start+4026531828>: 0x1badb002
  9     0x10002f:    mov    $0x0,%ebp       0x00000000      0xe4524ffe      0x7205c766
 10     0x100034:    mov    $0xf0117000,%e  0xf0100010 <entry+4>:   0x34000004      0x7000b8
 11     0x100039:    call   0x10009d        12      0x220f0011      0xc0200fd8
**    0x10000c: movw   $0x1234,0x472 (1000  (gdb)
```

图 1.16: 执行 movl %eax, %cr0 后的两处内存中的内容

可以看出，在执行了 movl %eax, %cr0 指令以后，两处的内容完全相同。这说明了此时已经成功启用了页表并完成了地址映射。

下面注释掉 movl %eax, %cr0，make clean 之后重新编译运行。逐步使用 stepi 之后在一个 jmp 指令后出现了问题，如图1.17。0x10002a 处的 jmp 指令需要跳转到 0xf010002c 处，但是没有分页管理不会进行虚拟地址映射到物理地址的转化，因此访问超出内存限制后程序崩溃。

```
66          # C code.                          ⇒ 0×10001d:    mov    %cr0,%eax
67          mov     $relocated, %eax          0×0010001d in ?? ()
68          jmp     *%eax                     (gdb)
69  relocated:                                ⇒ 0×100020:    or     $0×80010001,%eax
70                                            0×00100020 in ?? ()
71          # Clear the frame pointer registe (gdb)
72          # so that once we get into debugg ⇒ 0×100025:    mov    $0×f010002c,%eax
73          # stack backtraces will be termin 0×00100025 in ?? ()
74          movl    $0×0,%ebp                 (gdb)
75                                            ⇒ 0×10002a:    jmp    *%eax
76          # Set the stack pointer           0×0010002a in ?? ()
77          movl    $(bootstacktop),%esp      (gdb)
78                                            ⇒ 0×f010002c <relocated>:      add    %al,(%
79          # now to C code                   eax)
80          call    i386_init                 relocated () at kern/entry.S:74
81                                            (gdb)
82          # Should never get here, but in c Remote connection closed
/home/husixu/Homework/jOS/lab1/kern/entry.S (gdb)
```

图 1.17: jmp 指令后程序崩溃

Exercise 7 实验过程

1.3.2 Formatted Printing to the Console

阅读 kern/printf.c，lib/printfmt.c 和 kern/console.c 的代码，分析他们之间的关系。

Exercise 8

有一小部分用于打印八进制数（“%o” 格式）的代码被遗漏了，找出并补全这部分程序。

Exercise 8 实验过程

要进行这个练习首先要分析 kern/printf.c，kern/console.c 以及 lib/printfmt.c 之间的关系。初步观察 3 个文件，发现 kern/printf.c 中的 cprintf, vprintf 函数调用了 lib/printfmt.c 中的 vprintfmt 函数。kern/printf.c 中的 putch 程序和 lib/printfmt 程序调用了 kern/console.c 中的 cputchar 程序。进一步使用静态分析工具进行分析，cprintf 的调用关系图（call graph）如图1.18所示。

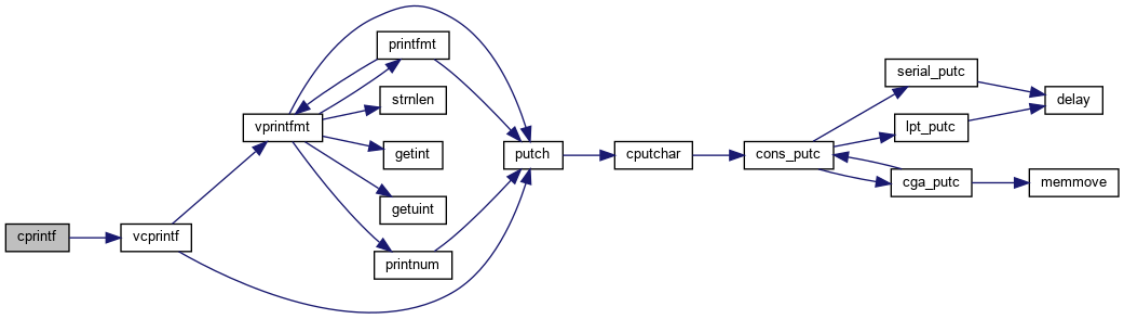

图 1.18: cprintf 的调用关系图

首先可以看到，这些函数中最接近底层的是 cons_putc，通过它的注释可以看到 cons_putc 的作用是输出一个字符到控制台。而 cons_putc 是通过调用 serial_putc，lpt_putc 以及 cga_putc 实现的。serial_putc 的代码及有关常量定义如下：

```
#define COM1          0x3F8
#define COM_LSR       5    // In:    Line Status Register
#define COM_LSR_TXRDY 0x20 //   Transmit buffer avail
#define COM_TX        0    // Out: Transmit buffer (DLAB=0)

static void serial_putc(int c) {
    int i;
    for (i = 0; !(inb(COM1 + COM_LSR) & COM_LSR_TXRDY) && i < 12800; i++)
        delay();
    outb(COM1 + COM_TX, c);
}
```

通过查询端口功能[4]，并分析代码，发现代码!(inb(COM1 + COM_LSR)& COM_LSR_TXRDY)是用于检测 03FD 的 bit5 是否为 1，而这一位用于判断发送数据缓冲区寄存器是否为空。outb 则是向 03F8 发送数据 c 而 03F8 被写入时功能为发送数据到串口。因此 serial_putc 的作用是向串口发送一个字符。

通过同样的方法，可以判断处 lpt_putc 的作用是向并口发送这个字符。最后，cga_putc 的作用是向 cga 设备，也就是计算机的显示器发送一个字符。cga_putc 对于需要输出的字符进行判断，如果是如'\t'、'\b' 等特殊字符则移动光标的位置或者执行相应动作，否则直接将字符显示输出在屏幕上。switch 后面的 if 用于判断缓冲区显示的内容不能够超过显示器大小 CRT_SIZE。

向上分析，cputchar 直接调用 cons_putc，而 putchar 直接调用 cputchar，并将计数器加一计数。vprintfmt 则调用 cputc 进行打印输出。printfmt 则是帮助 vprintfmt 进行递归调用的一个帮助函数。观察 vprintfmt，发现其由 4 个参数:

1. void (*putch)(int, void*) 是一个输出一个字符的函数指针。其第二个地址是字符输出位置的地址的地址，由于需要实现将值输出到这个地址后地址 +1，因此这个函数指针的第二个参数不是地址而是地址的地址。

2. void *putdat 是这个字符要存放的内存地址指针

3. const char* fmt 是输入的格式化字符串

4. va_list ap 是格式化字符串的可变长参数。

这个 vprintfmt 函数就是需要被修改的函数，将输出八进制的部分修改为如下代码即可实现 8 进制输出。

```
    case 'o':
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

修改完成以后重新编译，即在命令行中执行 make clean && make && make qemu 以后，qemu 启动后会出现一行说明修改成功的输出:6828 decimal is 15254 octal!，如图1.19所示。

[4] http://bochs.sourceforge.net/techspec/PORTS.LST

```
Machine  View
Booting from Hard Disk...
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
```

图 1.19: 重新编译并启动 qemu 后的部分输出

Exercise 8 实验过程

接下来对于实验指导中 Exercise 8 之后的问题，给出解答。

1. 解释 printf.c 和 console.c 之间的接口，尤其是 console.c 导出了那个函数？这个函数是如何在 printf.c 中被使用的?

   - 在 Exercise 8 中已经给出分析部分。console.c 中除了静态函数之外所有的函数都导出并为外部提供服务了。printf.c 调用 console.c 中的 cputchar 函数进行显示输出。

2. 解释如下 console.c 中的代码:

```
1  if (crt_pos >= CRT_SIZE) {
2      int i;
3      memmove(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  }
```

   - 这部分代码是为了当输出超过显示屏的大小的时候将显示屏向上滚动一行，由于显示屏大小为 80×25，因此将 1~24 行复制并放在 0~23 行，然后将最后一行清空。mmove 的作用是滚动，而 for 循环的作用是清空最后一行。

3. 单步执行如下代码:

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

   (a) 在调用 cprintf 的过程中，fmt 指向那里？ap 指向哪里?

   (b) 按照执行的顺序列出对于 cons_putc, va_arg, 以及 vcprintf 的调用。对于 cons_putc，列出它的参数。对于 va_arg，列出调用前后 ap 指向哪里。对于 vcprintf 列出它的两个参数的值。

   - 在 kern/monitor.c 中的 mon_backtrace 中添加这两条语句并打开 gcc 进行追踪，发现 fmt 指向格式化字符串”x %d, y%x, z%n”，而 ap 指向参数 x, y, z，如图1.20所示。(由于在 cprintf 中不能正常看到 fmt 以及 ap 的值因此实际取值实在 vcprintf 中进行的。

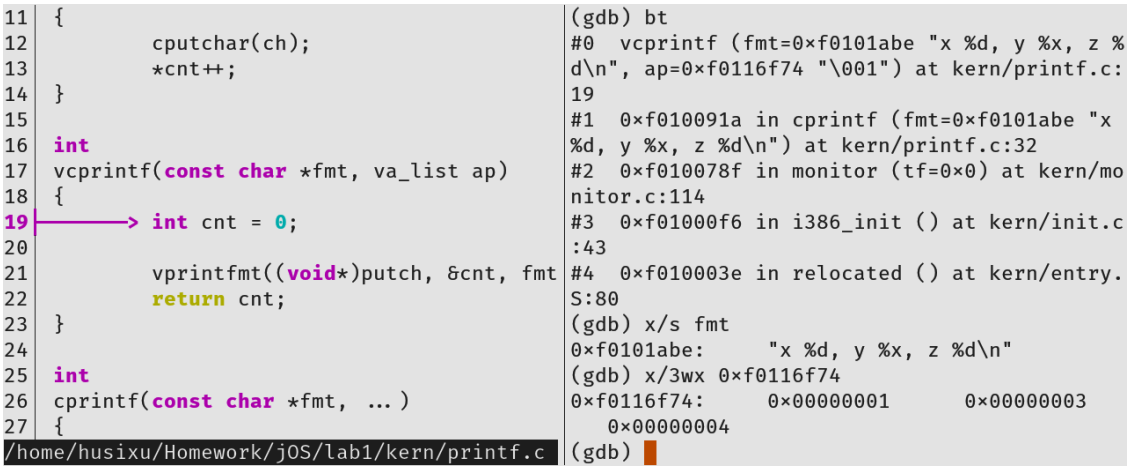

图 1.20: fmt 以及 ap 的取值

- 通过对于代码的跟踪，可以得知按照顺序，对于三个函数的调用过程如表1.2所示。
- 对于这三个函数的调用，getint 使用 va_arg 根据不同的参数类型从参数列表中取出下一个参数。第一次对于 va_arg 进行调用前 ap 为 1,3,4，调用后变为 3,4。最后，所有函数的调用顺序如表1.2所示。

表 1.2: vcprintf, va_arg 以及 cons_putc 的调用顺序

#	函数	备注
1	vcprintf	fmt=“x %d, y %x, z %d \n”, ap=1,3,4
2	cons_putc	c=120
3	cons_putc	c=32
4	va_arg	调用前 ap=1,3,4 调用后 ap=3,4
5	cons_putc	c=49
6	cons_putc	c=44
7	cons_putc	c=32
8	cons_putc	c=121
9	cons_putc	c=32
10	va_arg	调用前 ap=3,4 调用后 ap=4
11	cons_putc	c=51
12	cons_putc	c=44
13	cons_putc	c=32
14	cons_putc	c=122
15	cons_putc	c=32
16	va_arg	调用前 ap=4 调用后 ap 为空
17	cons_putc	c=52
18	cons_putc	c=10

4. 运行如下代码：

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

输出是什么？按照上一题的方法逐步解释输出是怎么来的。输出依赖于 x86 系统是小端这一事实。如果 x86 是大端的，当 i 为何值时才会有相同的输出？你需要将 57616 更改为一个不同的值吗？

- 与上一题类似，首先，将这两句代码加入 monitor.c 中，然后重新编译并运行，最后得到的运行结果为 “He110, World”，如图1.21所示。

```
Machine  View
SeaBIOS (version 1.11.0-20171110_100015-anatol)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92510+07EF2510 C980


Booting from Hard Disk...
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
He110 World
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

图 1.21: 重新编译后的运行结果

输出这个值的原因是：第一个%x 要求按照 16 进制输出第一个参数，而第一个参数的值为 57616，对应的 16 进制为 e110，因此前半部分输出为 He110，而第二个格式化字符%s 要求将 &i 作为一个字符串输出，此时将 i 进行拆分。由于 x86 是小端模式，因此地址由低向高增长时字节也由 LSB 向 MSB 增长。因此将 0x00646c72 拆分后，变为 0x72('r'), 0x6c('l'), 0x54('d'), 0x00('\0')。于是后半部分输出 World。当 i 是大端时，需要能够将 i 改为 0x726c6400 才能有相同的输出，但是不需要改变 57616 这个值。

5. 在接下来的代码中，'y=' 之后打印的是什么？（注意：答案不是一个特定值。）这为什么会发生？

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

- 同样的，将这个代码输入 monitor.c 后重新编译运行，输出的结果为 “x=3 y=-267292872”，x 后面输出的为 3，因为第一个参数为 3，而 y 没有指定参数，因此将输出一个不确定的值，实际上，是从栈中 3 这个参数的后方多取了一个数作为参数。

6. 让我们假设 GCC 改变了函数调用转换，并将参数入栈顺序更改为声明顺序，也就是说最后一个参数最后入栈。你需要如何更改 cprintf 或它的接口才能使传递任意参数仍然可行？

- 将变长参数置于格式化字符串的前方即可，即将 cprintf 的接口变为 `int cprintf( ... , const char *fmt)`。

1.3.3 The Stack

这一份将重新编写一个 kernel monitor 程序用于记录堆栈的变化，该变化是由一系列被保存到堆栈的 IP 寄存器的值组成的。

Exercise 9

判断操作系统的内核从哪一条指令开始初始化它的堆栈空间，以及这个堆栈空间在内存的哪个地方？内核是如何给堆栈保留一块内存空间的？堆栈指针是指向这个区域的哪一端的？

Exercise 9 实验过程

在进入内核之前，整个 boot loader 是没有对%esp 和%ebp 的内容进行修改的，因此这其中没有初始化堆栈空间的语句。而在 entry.S 中，发现其最后几条指令如下：

```
1  relocated:
2      # Clear the frame pointer register (EBP)
3      # so that once we get into debugging C code,
4      # stack backtraces will be terminated properly.
5      movl    $0x0,%ebp           # nuke frame pointer
6
7      # Set the stack pointer
8      movl    $(bootstacktop),%esp
9
10     # now to C code
11     call    i386_init
```

通过最后一条指令跳转到 C 函数中，以及前两条语句的注释可以判断，movl $0x0,%ebp 以及 movl $(bootstacktop),%esp 就是用于初始化堆栈空间的语句。这两条语句中，第一条将 0 赋值给%ebp，第二条将 $(bootstacktop) 赋值给%esp，此时需要进入 gdb 进行调试。当执行到这两条指令时，gdb 显示如图1.22所示。

```
74          movl    $0x0,%ebp                    Breakpoint 2, relocated () at kern/entr
75                                               y.S:74
76          # Set the stack pointer              (gdb) si
77  ------> movl    $(bootstacktop),%esp         => 0xf0100034 <relocated+5>:    mov
78                                               $0xf0117000,%esp
79          # now to C code                      relocated () at kern/entry.S:77
/home/husixu/Homework/jOS/lab1/kern/entry.S      (gdb)
```

图 1.22: gdb 显示输出

从图中可以发现，movl $(bootstacktop),%esp 在运行时为 mov $0xf0117000,%esp。也就是说栈顶的地址为 0xf0117000。而根据 entry.S 最后的标志可以发现，在栈顶前分配了 KSTKSIZE 这么多的空间。KSTKSIZE 在 inc/memlayout.h 中定义。其大小为 (8*PGSIZE)，也就是 32KB，因此栈的地址空间为 0xf010f000~0xf0117000。

综上所述，对于 Exercise 9 提出的问题进行回答：

1. 内核从哪一条指令开始初始化它的堆栈空间？

   - 从 movl $0x0,%ebp 以及 movl $(bootstacktop),%esp 开始初始化堆栈空间。

2. 这个堆栈空间在内存的哪个地方?
   - 虚拟地址 0xf010f000~0xf0117000，对应的物理地址 0xf010f000~0x00117000。
3. 内核是如何给堆栈保留一块内存空间的?
   - 通过在 entry.S 中的数据段声明一块大小为 32KB 的空间作为堆栈使用。
4. 堆栈指针是指向这个区域的哪一端的?
   - 由于栈是向下增长的，故堆栈指指向最高地址。最高地址就是 bootstacktop 的值 0xf0117000。

Exercise 9 实验过程

X86 堆栈指针寄存器%esp 指向堆栈中正在被使用的部分的最低地址。更低的地址空间还没有被利用。在 32 位模式中，每一次对堆栈的操作都是以 32bit 为单位的，因此%esp 是 4 字节对齐的。%ebp 寄存器是记录每一个程序的栈帧的相关信息的寄存器，每一个程序运行时都会分配给一个栈帧，用于存放临时变量，传递参数等。

Exercise 10

找到 obj/kern/kernel.asm 中 test_backtrace 子程序的地址，设置断点并讨论启动内核后这个程序被调用时发生了什么。对于递归的调用，多少 32 位字被压入栈中？这些字分别是什么?

Exercise 10 实验过程

在 obj/kern/kernel.asm 中，test_backtrace 所对应的子程序在开始时对应如下几条指令:

```
1  f0100040:   55                   push   %ebp
2  f0100041:   89 e5                mov    %esp,%ebp
3  f0100043:   53                   push   %ebx
4  f0100044:   83 ec 14             sub    $0x14,%esp
```

这 5 条指令用于保存父程序的栈信息，并为当前子程序分配新的栈帧。在 i386_init 中运行了子程序 test_backtrace(5)。在调用这一子程序之前，%esp = 0xf010ffe0, %ebp = 0xf010fff8，如图1.23所示。

```
36          cprintf("6828 decimal is %o octal!\n",
37
38          // Test the stack backtrace function (l
39 ------>  test_backtrace(5);
40
41          // Drop into the kernel monitor.
/home/husixu/Homework/jOS/lab1/kern/init.c
```

```
=> 0xf01000de <i386_init+65>:   movl
$0x5,(%esp)
(gdb) p $esp
$2 = (void *) 0xf0116fe0
(gdb) p $ebp
$3 = (void *) 0xf0116ff8
(gdb)
```

图 1.23: 调用 test_backtrace(5) 前的栈寄存器信息

在调用过程中，首先 call 指令将 i386_init 的返回地址压入栈中，此时 esp 变为 0xf010ffdc，此时进入 test_backtrace(5) 子程序。然后开始执行上述的 5 个指令：push %ebp 用于保存%ebp 寄存器的值，执行完成后%esp 变为 0xf010ffd8；然后 mov %esp, %ebp 把%ebp 的值更新为%esp 的值，也就是 0xf010ffd8；然后 push %ebt 保护寄存器%ebx 的值，执行完成后 esp 变为 0xf010ffd4；最后 sub $0x14,

%esp 将%esp 减去 0x14，%esp 变为 0xf010ffc0，这是用于分配空间存储临时变量。因此，在执行完上述指令后，%esp 和%ebp 分别变为 0xf010ffc0 和 0xf010ffd8。而这就是 test_backtrace(5) 执行时栈寄存器的变化范围。由于递归调用，像这样的变化还需要执行 test_backtrace(4)~test_backtrace(0) 5 次。

也就是说每次递归调用压入 32 字节，也就是 8 个 32 位字。在调用到 test_backtrace(0) 时，%esp 将为 0xf010ff20 而%ebp 将为 0xf010ff38。实际的调试过程也验证了这一点，如图1.24所示。

```
11  void                                                    [88/88]
12  test_backtrace(int x)                  Breakpoint 3, test_backtrace (x=0) at k
13  {                                      ern/init.c:14
14  ------> cprintf("entering test_backtrace %d\n",  (gdb) p $esp
15          if (x > 0)                     $4 = (void *) 0×f0116f20
16                  test_backtrace(x-1);   (gdb) p $ebp
17          else                           $5 = (void *) 0×f0116f38
/home/husixu/Homework/jOS/lab1/kern/init.c (gdb)
```

图 1.24: 执行到 test_backtrace(0) 时的栈寄存器值

Exercise 10 实验过程

上述练习已经有足够多的信息以供实现一个 stack backtrace 程序了，将之称为 mon_backtrace()，其原型已经定义在 kern/monitor.c 中。除了实现之外，好需要将其 hook 到 kernel monitor 命令列表中，使其能够直接被用户调用。

backtrace 函数应该显示一个如下的调用帧关系：

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

每一行都包含 ebp, eip 和 args，也就是有关的寄存器以及参数。而第一行的 Stack backtrace 表明现在执行的是 mon_backtrace 子程序。其后依次外推直到最外层。

Exercise 11

实现上述 backtrace 程序，并使用相同的格式进行输出。使用 make grade 进行验证。

Exercise 11 实验过程

根据程序调用时栈的关系来实现显示正在执行的程序的栈的信息，栈的结构如图1.25所示。

可以看出，保存的%ebp 是连接当前栈帧和上一个（调用者的）栈帧的关键，它使各个栈帧通过类似于链表的结构连接起来。最终，在 mon_backtrace 中填写的代码如下：

```
1  int mon_backtrace(int argc, char **argv, struct Trapframe *tf) {
2      uint32_t ebp, eip;
3      cprintf("Stack backtrace:\n");
4      for (ebp = read_ebp(); ebp != 0; ebp = *((uint32_t *)ebp)) {
5          eip = *((uint32_t *)ebp + 1);
6          cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n",
7              ebp, eip, *((uint32_t *)ebp + 2),
8              *((uint32_t *)ebp + 3), *((uint32_t *)ebp + 4),
9              *((uint32_t *)ebp + 5), *((uint32_t *)ebp + 6));
```

```
10        }
11        return 0;
12    }
```

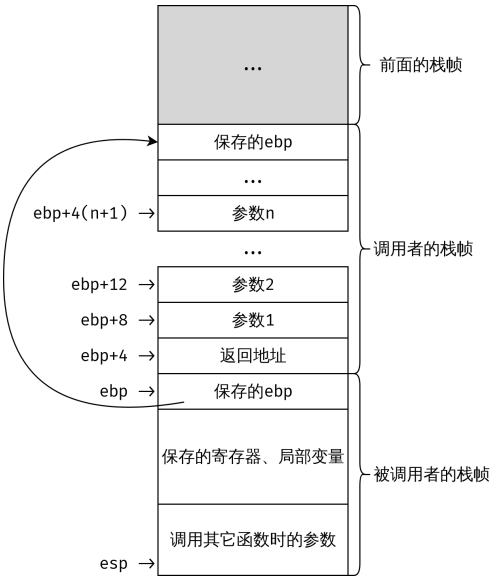

图 1.25: 栈帧结构

使用 make grade 进行测试，结果如图1.26所示。从结果中可以看出，backtrace count 以及 backtrace arguments 已经通过测试，余下的 backtrace arguments 和 backtrace lines 需要在 exercise 12 中进一步补全。

```
running JOS: (1.6s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: FAIL
    AssertionError: got:

    expected:
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      i386_init

  backtrace lines: FAIL
    AssertionError: No line numbers

Score: 40/50
```

图 1.26: 测试结果

Exercise 11 实验过程

Exercise 12

继续修改 backtrace 函数，令其对于每个 eip 能显示函数名，源文件以及对应的行数。

在 debuginfo_eip 中，___STAB_* 是从哪来的？这个问题的答案很长，可能需要以下操作来帮助找到答案：

- 看看 kern/kernel.ld 中有哪些 ___STAB_*
- 运行 objdump -h obj/kern/kernel
- 运行 objdump -G obj/kern/kernel
- 运行 gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c，然后查看 init.s
- 看看 bootloader 是否将符号表作为内核的一部分加载到了内存中。

继续完成 debuginfo_eip 的实现，通过调用 stab_binsearch 来查找地址对应的行数。

在 kernel monitor 中添加 backtrace 命令，并扩展 mon_backtrace 函数，令其调用 debuginfo_eip 然后对于每一个栈帧打印如下格式的一行：

```
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386\_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
```

每一行都给出了文件名以及行号，紧跟着的是函数名以及 eip 相对于这个函数第一条指令的偏移。

Exercise 12 实验过程

stab 表是用于存储源文件以及编译后文件对应关系的一张表。使用 objdump -G obj/kern/kernel 命令，查看 stab 表中的内容，其输出如图1.27所示。

```
13     SLINE  0      83     f010003e 0
14     SO     0      2      f0100040 31     kern/entrypgdir.c
15     OPT    0      0      00000000 49     gcc2_compiled.
16     LSYM   0      0      00000000 64     int:t(0,1)=r(0,1);-2147483648;2147483647;
17     LSYM   0      0      00000000 106    char:t(0,2)=r(0,2);0;127;
18     LSYM   0      0      00000000 132    long int:t(0,3)=r(0,3);-2147483648;2147483647;
19     LSYM   0      0      00000000 179    unsigned int:t(0,4)=r(0,4);0;4294967295;
20     LSYM   0      0      00000000 220    long unsigned int:t(0,5)=r(0,5);0;4294967295;
21     LSYM   0      0      00000000 266    long long int:t(0,6)=r(0,6);-0;4294967295;
22     LSYM   0      0      00000000 309    long long unsigned int:t(0,7)=r(0,7);0;-1;
23     LSYM   0      0      00000000 352    short int:t(0,8)=r(0,8);-32768;32767;
```

图 1.27: stab 表部分内容

图中，前五列对应的就是 stab，第 7 列是 stabstr，分别对应着 kdebug.c 中的 ___STAB_BEGIN___

以及 ___STABSTR_BEGIN___。

通过 stab_binsearch 对于 stab 进行检索。通过注释可以知道 stab_binsearch 接受 4 个参数：

- const struct Stab *stabs：stab 表的起始位置。
- int *region_left：查找下界。
- int *region_right：查找上界。
- int type：查找的表项类型，对应图1.27的第二列
- uintptr_t addr 要查找的值，对应图1.27的第五列。

通过这个函数对于 stab 表进行查找，可知在 debuginfo_eip 中需要补全的部分为：

```
1 stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
2 info->eip_line = lline > rline ? -1 : stabs[rline].n_desc;
```

将 debuginfo_eip 补全以后，需要对于 kern/monitor.c 中的 mon_backtrace 进行扩展，使其调用这一函数以获得相应的调试信息。扩展后的 mon_backtrace 如下：

```
1  int mon_backtrace(int argc, char **argv, struct Trapframe *tf) {
2      uint32_t ebp, eip;
3      struct Eipdebuginfo info;
4      cprintf("Stack backtrace:\n");
5      for (ebp = read_ebp(); ebp != 0; ebp = *((uint32_t *)ebp)) {
6          eip = *((uint32_t *)ebp + 1);
7          debuginfo_eip(eip, &info);
8          cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n      %s
               :%d: %.*s+%d\n",
9              ebp, eip,
10             *((uint32_t *)ebp + 2), *((uint32_t *)ebp + 3), *((uint32_t *)ebp +
                   4),
11             *((uint32_t *)ebp + 5), *((uint32_t *)ebp + 6),
12             info.eip_file, info.eip_line, info.eip_fn_namelen,
13             info.eip_fn_name, eip - info.eip_fn_addr);
14     }
15     return 0;
16 }
```

然后重新编译，运行 make grade 的结果如图1.28所示，可以看到已经通过全部的测试。

最后，需要将 backtrace 命令整合进入终端中，这时只需将 kern/monitor.c 中的 commands 更改为如下几行即可。

```
1 static struct Command commands[] = {
2     { "help", "Display this list of commands", mon_help },
3     { "kerninfo", "Display information about the kernel", mon_kerninfo },
4     { "backtrace", "Display information about function call frames",
          mon_kerninfo },
5 };
```

```
running JOS: (1.1s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
 ~/H/jos  lab1 *…  lab1  2403ms  Sat Feb 24 00:10:06 2018
```

图 1.28: 重新编译后运行 make grade 结果

更改后重新编译并进入 qemu 进行测试，可以看到，输入 help 后能够显示 backtrace 命令，并且输入 backtrace 后能够正确的输出结果了，如图1.29所示。

```
Machine  View
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Display information about function call frames
K> backtrace
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101842 (virt)  00101842 (phys)
  edata  f011a300 (virt)  0011a300 (phys)
  end    f011a944 (virt)  0011a944 (phys)
Kernel executable memory footprint: 107KB
```

图 1.29: 将 backtrace 整合到终端中

Exercise 12 实验过程

第 2 章　Memory Management

2.1　Physical Page Management

操作系统需要记录哪些内存区域是空闲的，哪些是正在被使用的。jOS 使用页为最小粒度来管理 PC 的物理内存区域，使其可以使用 MMU 对分配的内存进行映射和保护。

下面需要编写一个物理内存分配器：它使用一个 struct PageInfo 的链表来跟踪哪些内存是空闲的。每个结构对应一个物理页。此物理内存分配工具是其他的虚拟内存工具的基础。

Exercise 1

在 kern/pmap.c 中实现以下的函数：

- boot_alloc()
- mem_init()
- page_init()
- page_alloc()
- page_free()

check_page_free_list() 和 check_page_alloc() 可以用于检查分配器实现是否正确。通过启动 jOS 来调用 check_page_alloc()

Exercise 1 实验过程

首先通过观察 kern/init.c，发现内核在初始化的时候会调用 meminit 函数，而 mem_init() 在调用 i386_detect_memory 函数检测系统中有多少可用的内存空间之后就会调用 boot_alloc 函数进行空间分配。通过 boot_alloc 函数的注释可知，这个函数只是暂时用于被当做页分配器，而使用的真实也分配器是 page_alloc 函数。boot_alloc 只维护一个局部静态变量 nextfree 用于存放下一个可以使用的空闲空间的虚拟地址。所以每次需要分配 n 个字节的内存的时只需要修改这个值即可。除了修改这个值之外还需要判断是否有足够的内存可以分配。因此 boot_alloc 添加的代码如下。在 boot_alloc(PGSIZE); 执行完成后，也就分配了一个页的内存。

```
1  result = nextfree;
2  nextfree = ROUNDUP(nextfree + n, PGSIZE);
3  if((uint32_t)nextfree - KERNBASE > (npages * PGSIZE))
4      panic("boot alloc: out of memory.\n");
```

```
5  return result;
```

上述命令执行完成后，下一条更改 kern_pgdir 的指令为页目录添加一个目录表项。UVPT 是一段虚拟地址的起始地址，而 PADDR 取的是 kern_pgdir 的物理地址，而这一条指令也就是将虚拟地址映射到物理地址。根据注释可知，对于内核和用户而言这段内存都是只读的。

接下来就是需要补充完整的 mem_init 的部分。通过注释得知这一部分需要分配一块内存用于存放一个 PageInfo 数组，数组中的每一个 PageInfo 用于记录内存中的一页。同样，使用 boot_alloc 完成这一部分，这个数组的大小应该 $npages \times \text{sizeof}(PageInfo)$。补全的代码如下：

```
1  pages = (struct PageInfo *)boot_alloc(npages * sizeof(struct PageInfo));
2  memset(pages, 0, npages * sizeof(struct PageInfo));
```

接下来 mem_init 将执行 page_init 函数，也是下一个需要补全的函数。同样，通过注释可以发现这个函数的功能有两个：初始化业庙的结构以及 pages_free_list，也就是空闲页的信息。在这个函数调用完成后 boot_alloc 将不再被使用。

根据注释中的信息，得知第 0 页、IO hole、以及 extended memory 中的内核部分和一些其它部分已经被占用，因此，最终修改得到的代码如下：

```
1  size_t i;
2  const size_t pages_in_use_end =
3      npages_basemem + 96 + ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;
4  // set page 0 in use
5  pages[0].pp_ref = 1;
6  // set low memory
7  for (i = 1; i < npages_basemem; ++i){
8      pages[i].pp_ref = 0;
9      pages[i].pp_link = page_free_list;
10     page_free_list = &pages[i];
11 }
12 // set IO hole
13 for (i = npages_basemem; i < pages_in_use_end; ++i){
14     pages[i].pp_ref = 1;
15 }
16 // set extended memory
17 for (i = pages_in_use_end; i < npages; ++i) {
18     pages[i].pp_ref = 0;
19     pages[i].pp_link = page_free_list;
20     page_free_list = &pages[i];
21 }
```

处理与物理内存页有关的数据之后执行的 check_page_free_list(1) 以及 check_page_alloc() 是对于已经分配的页表的检查，查看页表以及空闲页是否为合法，以及检查 page_alloc 以及 page_free 是否能够正确运行。接下来就应该实现 page_alloc 以及 page_free 函数了。

通过 page_alloc 的注释可以知道这个函数的功能是分配一个物理页，并返回这个物理页所对应的 PageInfo 结构，其主要的工作是对于 free_page_list 进行更改。完成后的 page_alloc 代码如下：

```
1  struct PageInfo * page_alloc(int alloc_flags) {
2      struct PageInfo *temp;
3      if (!page_free_list)
4          return NULL;
5      temp = page_free_list;
6      page_free_list = temp->pp_link;
7      temp->pp_link = NULL;
8      if (alloc_flags & ALLOC_ZERO)
9          memset(page2kva(temp), 0, PGSIZE);
10     return temp;
11 }
```

首先检查 page_free_list 中是否还有空闲的节点，如果有则将其取出进行分配，否则返回 NULL。然后根据注释的要求，如果 alloc_flags & ALLOC_ZERO 为真，则将这一页内容全部置零。

接下来实现 page_free。同样，根据注释，要实现这个函数就是要将一个 PafeInfo 重新连接到 page_free_list 之中，代表对于这个页的回收。因此实现的代码如下。根据注释提示，当 pp_ref 或者 pp_link 不为 0 时应该发出 panic。

```
1  if(pp->pp_ref || pp->pp_link)
2      panic("page_free: illegal PageInfo.");
3  pp->pp_link = page_free_list;
4  page_free_list = pp;
```

至此对于物理内存管理的代码已经补充完整。注释掉 mem_init 中的第一个 panic 并重新编译、运行 qemu，得到的输出如图2.1所示。可以看到，check_page_free_list 和 check_page_alloc 的检查已经通过。而最后的 check_page 检查需要在下一个实验中完成。

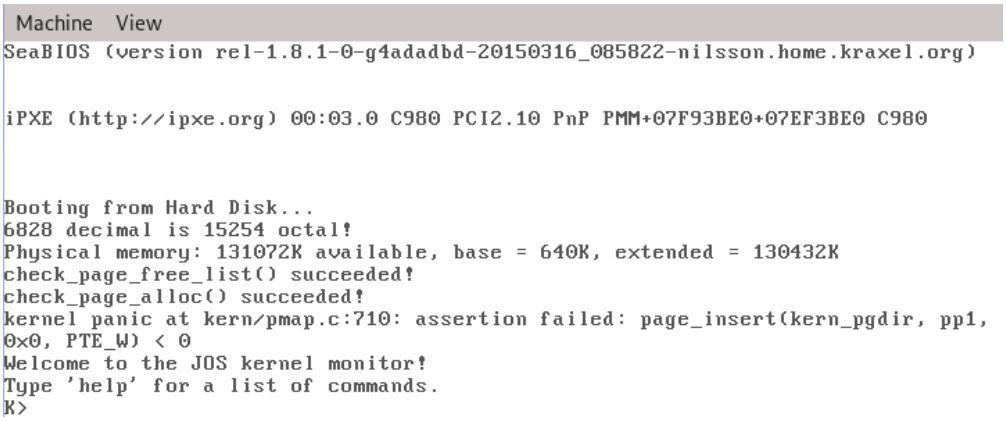

图 2.1: 重新编译后运行 qemu 的输出

Exercise 1 实验过程

2.2 Virtual Memory

Exercise 2

阅读 *Intel 80386 Reference Manual*[a]的第 5 章和第 6 章，并仔细阅读有关分页转换以及基于分页的保护。

[a] https://pdos.csail.mit.edu/6.828/2017/readings/i386/toc.htm

Exercise 2 实验过程

对于分页转换而言，需要注意的是线性地址的格式、页表的格式以及线性地址如何转换为物理地址。线性地址的 31~22bit 为页目录地址，21~12 为页地址，而 11~0 位为偏移量。线性地址到物理地址的转换过程如图2.2所示。

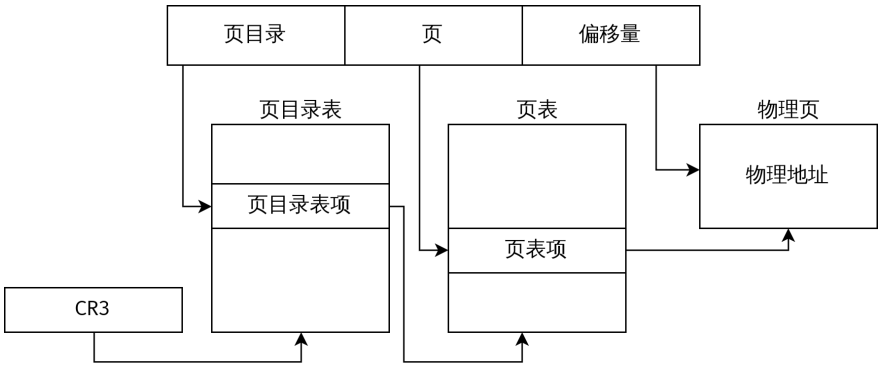

图 2.2: 线性地址到物理地址的转换

而对于每一个页表项，其不仅包含物理页的地址，还包含一系列的表示位用于表示页的属性，如权限、使用标志等。基于分页的保护就是建立在这种页表项的基础之上的。基于分页的保护参数包括 2 个标志位，一个是页表项中的 U/S 位，当其为 0 时表示监督模式，此时这个分页用于操作系统以及系统软件，当其为 1 时表示用户模式，为用户态软件提供服务。另一个标志位则是 R/W 位，当其为 0 时表示只读模式，为 1 时则表示读写模式。

Exercise 2 实验过程

2.2.1 Virtual, Linear, and Physical Addresses

在 x86 系统中，一个虚拟地址由两部分组成：段选择器和段内偏移。线性地址值得是通过段地址转换机构把虚拟地址进行转换之后得到的地址，而物理地址则是分页地址转换机构将线性地址转换之后得到的真实内存地址。

C 语言中的指针是虚拟地址中的段内偏移部分。boot/boot.S 中的的全局描述符表将所有段的基址设置为 0，上限设置为 0xffffffff，因而关闭了分段管理的功能。此时虚拟地址中段选择器字段的内容失去了意义，因为线性地址的值总是等于虚拟地址中段内偏移的值。lab1 中 part3 引入了一个简单的页表，这个页表只映射了 4MB 的空间。在 jOS 中需要将这种映射扩展到 256MB 的空间上。

Exercise 3

Gdb 只能够通过虚拟地址访问 qemu 的内存，但如果能够访问到实际内存地址将会是十分有用的。在运行 qemu 的终端中按下 Ctrl-a c 即可进入 qemu 的监视器

在 qemu 监视器中使用 xp 命令以及 gdb 中的 x 命令来查看虚拟地址和对应物理地址的内存内容。对于 6.828 patch 版本的 qemu 而言，info pg 能够显示页表的内容，而 info mem 能够显示所有已经被页表映射的虚拟地址的空间以及它们的访问优先级。

Exercise 3 实验过程

编译并运行 qemu，在进入内核之后，在 GDB 中使用 x 命令检查 0xf0100000 处的部分内容，并使用 qemu 的 xp 命令检查 0x00100000 处的部分内容，其结果如图2.3所示。

```
18          // Before doin (gdb) x/5i 0×f0100000                                    (qemu) xp/5i 0×00100000
19          // Clear the u    0×f0100000 <_start+4026531828>:      add    0×1bad(%eax),%dh 0×00100000:  add    0×1bad(%eax),%dh
20          // This ensure    0×f0100006 <_start+4026531834>:      add    %al,(%eax)       0×00100006:  add    %al,(%eax)
21 ------> memset(edata,      0×f0100008 <_start+4026531836>:      decb   0×52(%edi)       0×00100008:  decb   0×52(%edi)
22                            0×f010000b <_start+4026531839>:      in     $0×66,%al        0×0010000b:  in     $0×66,%al
23          // Initialize     0×f010000d <entry+1>:        movl   $0×b81234,0×472          0×0010000d:  movl   $0×b81234,0×472
/home/husixu/Homework/jOS/ (gdb)                                                        (qemu)
```

图 2.3: qemu 以及 gdb 对于映射地址内容的显示

从图中可以看出，映射后的虚拟地址的内容和物理地址的内容是一致的。在 qemu 中输入 info pg 以及 info mem 后其输出如图2.4所示。从图中可以看出，qmeu 打印出了页表的内容以及映射的虚拟地址空间。

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
  [00000-000ff]  PTE[000-0ff] --------WP 00000-000ff
  [00100-00100]  PTE[100]     ----A---WP 00100
  [00101-00119]  PTE[101-119] --------WP 00101-00119
  [0011a-0011a]  PTE[11a]     ---DA---WP 0011a
  [0011b-003ff]  PTE[11b-3ff] --------WP 0011b-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
  [f0000-f00ff]  PTE[000-0ff] --------WP 00000-000ff
  [f0100-f0100]  PTE[100]     ----A---WP 00100
  [f0101-f0119]  PTE[101-119] --------WP 00101-00119
  [f011a-f011a]  PTE[11a]     ---DA---WP 0011a
  [f011b-f03ff]  PTE[11b-3ff] --------WP 0011b-003ff
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
(qemu)
```

图 2.4: qemu 中 info pg 以及 info mem 的输出

Exercise 3 实验过程

进入保护模式之后，就不能够直接使用物理地址和线性地址了，所有代码中的地址都是虚拟地址的形式，并被 MMU 转换。jOS 内核需要同时操作虚拟地址和物理地址但不对其解引用。为了帮助记录代码，jos 中有两个类型的指针：unitptr_t 表示虚拟地址，而 physaddr_t 表示物理地址。实际上，它们都是 32 位整型值 (uint32_t)。正因为如此编译器不会阻止将一个类型赋值给另一个类型，但是会阻止对这些值的解引用。要对其解引用，需要首先将其强制转换为一个指针类型。

Question

1. 假设以下 jOS 内核代码是正确的，那么 x 应该是 unitptr_t 类型还是 physaddr_t 类型?

```
1 mystery_t x;
2 char* value = return_a_pointer();
3 *value = 10;
4 x = (mystery_t) value;
```

Answer

1. 由于使用了 * 操作符进行了解引用，因此变量 x 应该为虚拟地址，是 uintptr_t 类型。

Answer

2.2.2 Reference counting

在之后的实验中将会遇到多个不同的虚拟地址被同时映射到相同的物理页面上的情况。在这种情况下我们需要记录每一个物理页面上存在多少不同的虚拟地址来引用它。这个值存放在 PageInfo 的 pp_ref 中。当这个值变为 0 时物理页才可以被释放。通常情况下任意一个物理页 p 的 pp_ref 的值等于它所在的所有页表中被虚拟地址 UTOP 下方的虚拟页映射的次数。

2.2.3 Page Table Management

这一部分需要创建一个页表管理程序，包括插入和删除线性地址到物理地址的映射，以及页表的创建等。

Exercise 4

完成 kern/pmap.c 中以下几个函数的实现。

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

mem_init 中调用的 check_page 可以用于检查这些函数的实现是否正确。

Exercise 4 实验过程

首先实现 pgdir_walk。根据注释，可以得知该函数所需要实现的功能为：给定一个页表目录指针 pgdir，该函数应该返回线性地址 va 对应的页表项的指针。此时可能这个页不存在，当这个页不存在时

如果 create==true 则分配新的页，否则返回 NULL，也就是说实现函数时需要注意页目录项对应的页表是否存在于内存中。最终，函数的实现如下：

```
pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create) {
    pde_t *pgdir_entry = pgdir + PDX(va);
    if (!(*pgdir_entry & PTE_P)) {
        if (!create)
            return NULL;
        else {
            struct PageInfo *new_page = page_alloc(1);
            if (!new_page)
                return NULL;
            *pgdir_entry = (page2pa(new_page) | PTE_P | PTE_W | PTE_U);
            ++new_page->pp_ref;
        }
    }
    return (pte_t *)(KADDR(PTE_ADDR(*pgdir_entry))) + PTX(va);
}
```

接下来是 boot_map_region 函数。这个函数要求将虚拟空间 $[va, va + size)$ 映射到物理空间 $[pa, pa + size)$ 的这种映射关系加入到 pgdir 中，其中 size 是 PGSIZE 的整数倍大小。此函数主要是为了设置 UTOP 之上的静态映射地址范围。最终，函数的实现如下，函数中，每一轮循环将一个物理页和虚拟也的映射关系加入到页表中，直到将 size 个字节分配完。

```
static void boot_map_region(pde_t *pgdir, uintptr_t va,
        size_t size, physaddr_t pa, int perm) {
    int offset;
    pte_t *pgtable_entry;
    for (offset = 0; offset < size; offset += PGSIZE, va += PGSIZE, pa +=
        PGSIZE) {
        pgtable_entry = pgdir_walk(pgdir, (void *)va, 1);
        *pgtable_entry = (pa | perm | PTE_P);
    }
}
```

对于 page_lookup 函数而言，它返回 va 所映射的物理页的 PageInfo 指针，且若 page_store 不为 0 则将物理页的页表地址存入 pte_store 中，如果 va 没有被映射则返回 NULL。其实现如下：

```
struct PageInfo *page_lookup(pde_t *pgdir, void *va, pte_t **pte_store) {
    pte_t *pgtable_entry = pgdir_walk(pgdir, va, 0);
    if (!pgtable_entry || !(*pgtable_entry & PTE_P))
        return NULL;
    if (pte_store)
        *pte_store = pgtable_entry;
    return pa2page(PTE_ADDR(*pgtable_entry));
}
```

下面是 page_remove 函数，其作用是将虚拟地址 va 的映射关系删除。其中，pp_ref 的值要减 1，且如果 pp_ref 减为 0 则需要将这个页回收，最后，这个页的页表项需要被置为 0，且如果移走了这个页，TLB 需要被无效化。最终的 page_remove 实现如下：

```
1  void page_remove(pde_t *pgdir, void *va) {
2      pte_t *pgtable_entry;
3      struct PageInfo *page = page_lookup(pgdir, va, &pgtable_entry);
4      if (!page)
5          return;
6      page_decref(page);
7      tlb_invalidate(pgdir, va);
8      *pgtable_entry = 0;
9  }
```

最后，是对于 page_insert 的实现。其功能与 page_remove 相对，也就是将映射到虚拟内存中的 va 映射到物理内存页 pp 上。如果 va 已经映射，则应该使用 page_remove 将其移除。并且页表应该按需分配并插入到 pgdir 中，且 pp_ref 需要加一，最后的函数实现如下：

```
1  int page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm) {
2      pte_t *pgtable_entry = pgdir_walk(pgdir, va, 1);
3      if (!pgtable_entry)
4          return -E_NO_MEM;
5      ++pp->pp_ref;
6      if ((*pgtable_entry) & PTE_P) {
7          tlb_invalidate(pgdir, va);
8          page_remove(pgdir, va);
9      }
10     *pgtable_entry = (page2pa(pp) | perm | PTE_P);
11     *(pgdir + PDX(va)) |= perm;
12     return 0;
13 }
```

将这些函数补充完整后，重新编译并启动 qemu。qemu 的输出结果如图2.5所示。

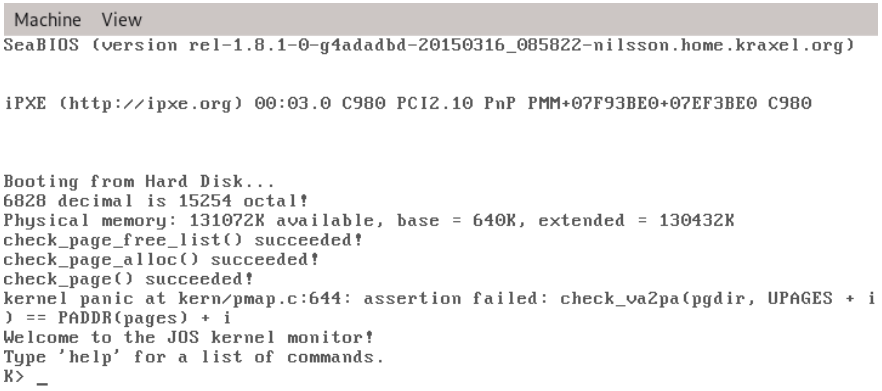

图 2.5: 重新编译运行后 qemu 的输出

从图中可以看到，check_page 的检查已经通过，说明上述 5 个函数的实现基本正确，接下来的 kernel panic 需要在下一个实验中解决。

Exercise 4 实验过程

2.3 Kernel Address Space

jOS 将处理器的线性地址划分为占用低地址的用户环境和占用高地址的内核，其界限是 inc/memlayout.h 中的变量 ULIM。jOS 为内核保留了 256M 的地址空间，这也就是为什么在 Lab 1 中要给操作系统设计一个非常高的地址空间。

2.3.1 Permissions and Fault Isolation

由于内核和用户进程只能访问各自的地址空间，所以我们不许在 x86 中使用权限位来将用户的代码限制在只能访问在用户空间可写权限位 PTE_W 可以同时影响内核和用户代码。高于 ULIM 的内存内核可读写但用户没有权限。用户和内核在 $[UTOP, ULIM)$ 有同样的可读不可写权限。低于 UTOP 的地址是用户空间。

2.3.2 Initializing the Kernel Address Space

在这一部分，需要设置 UTOP 之上的，给内核使用的空间。inc/memlayout.h 展示了应该使用的内存布局。

Exercise 5

将 mem_init() 内的 check_page() 之后的代码补充完整。可以使用 check_kern_pgdir() 以及 check_page_installed_pgdir() 来检查实现的代码。

Exercise 5 实验过程

根据 mem_init 中 check_page 之后的注释。首先，我们需要将 pages 数组映射到线性地址 UPAGES 上，此处使用 boot_map_region 函数进行映射，添加的代码为：

```
1  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
```

然后映射物理地址到内核的栈区域，将 bootstack 标记的物理地址映射到内核的堆栈。但是只需要映射 $[KSTACKTOP - KSTKSIZE, KSTACKTOP)$ 这部分区域。因此此处补充的代码为：

```
1  boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack),
       PTE_W);
```

最后，映射虚拟地址 $[KERNBASE, 2^{32})$ 到物理地址 $[0, 2^{32} - KERNBASE)$，内核的权限为读写，因此补充的代码为：

```
1  boot_map_region(kern_pgdir, KERNBASE, (0xffffffff-KERNBASE), 0, PTE_W);
```

最后，重新编译 jOS 并启动 qemu，其输出如图2.6所示。从图中可以看出，所有的测试已经通过。

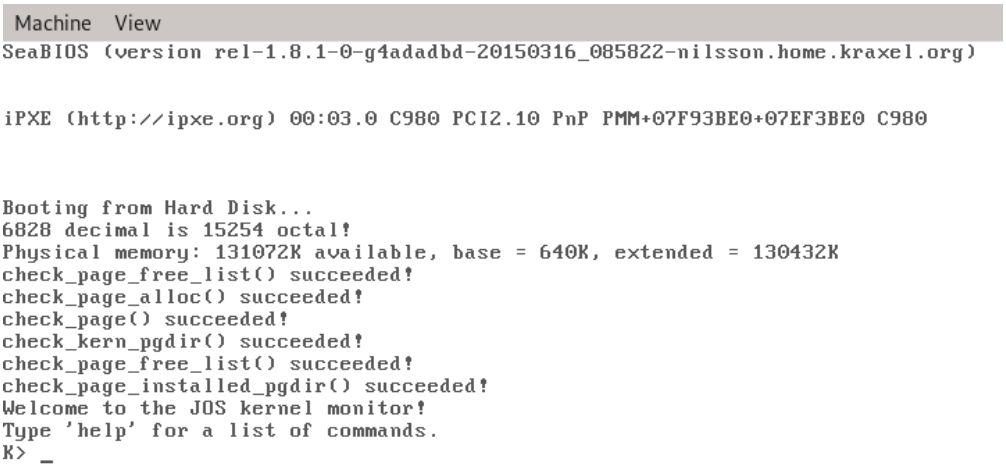

图 2.6: 重新编译并启动 qemu 后的输出结果

Exercise 5 实验过程

Question

2. 至此 page directory 被填入了哪些行？它们又被映射到了哪里？填写下面这张表。

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

3. 如果我们把 kernel 和 user environment 放在同一个地址空间中，为什么用户程序不能读写内核的内容？什么机制防保护了内核空间？

4. 这个操作系统最大可以支持多少内存？为什么？

5. 如果现在有最大的物理内存，那么我们需要多找额外的空间管理这些内存？这些额外的空间是怎么组成的？

6. 回顾 kern/entry.S 以及 kern/entrypgdir.c。在分页刚被打开时，EIP 仍然是一个很低的数（1MB 多一点）。在哪里 EIP 变得比 KERNBASE 高的?在刚开启分页到 EIP 比 KERNBASE 高的这段时间内，是什么让 EIP 保持一个很小的值但是程序继续运行的？为什么这种转换是必要的？

Answer

2. 通过 inc/memlayout.h，可以知道 page directory 实际如下：

Entry	Base Virtual Address	Points to (logically)
1023	0xffc00000	Page table for top 4MB of phys memory
...	...	...
960	0xf0000000	KERNBASE
959	0xefc00000	Kernel Stack
957	0xef400000	Page Directory
956	0xef000000	PageInfo array
...	...	...
0	0x00000000	Empty

3. 因为用户不能访问没有 PTE_U 权限的页，叶保护机制保护了内核空间。

4. 最大可以支持 4G 的空间。

5. 执行了一下代码以后 EIP 变得比 KERNBASE 高。

```
1 mov $relocated, %eax
2 jmp *%eax
```

之所以代码能够继续执行，是因为 0~4MB 的地址空间映射到了 0~4MB 的物理地址 (kern/entrypgdir.c) 中。转换的必要性在于需要在高地址处执行代码。

Answer

第 3 章　User Environment

3.1　User Environments and Exception Handling

inc/env.h 中包含了基本的环境定义。在 kern/env.c 中，可以看到内核维护了 3 个全局变量来存储环境。

- struct Env *envs = NULL; //ALL environments
- struct Env *curenv = NULL; The current env
- static struct Env *env_free_list; //Free environment list

jOS 开始运行以后，env 指针将指向一个存放系统各种环境的 Env 结构体数组。jOS 内核最大支持 NENV 个同时活动的环境。jOS 内核使用 env_free_list 维护所有不同的 Env 结构体，类似于空闲链表。内核使用 curenv 来表示当前运行的环境。内核启动前这个变量是 NULL。

3.1.1　Environment State

Env 结构体在 inc/env 中定义：

```
1  struct Env {
2      struct Trapframe env_tf;    // Saved registers
3      struct Env *env_link;       // Next free Env
4      envid_t env_id;             // Unique environment identifier
5      envid_t env_parent_id;      // env_id of this env's parent
6      enum EnvType env_type;      // Indicates special system environments
7      unsigned env_status;        // Status of the environment
8      uint32_t env_runs;          // Number of times environment has run
9
10     // Address space
11     pde_t *env_pgdir;           // Kernel virtual address of page dir
12 };
```

其中：

- env_tf：定义在 inc/trap.h 中，用于存放环境停止运行时寄存器的值。切换为内核模式的时候也会保存寄存器的值。

- env_link：指向 env_free_list 中的下一个 Env，env_free_list 空闲链表的第一个 env 环境。
- env_id：内核储存 env_id 环境的父用户环境 id。
- env_type：用来特定环境。
- env_status：这个变量可能为以下几种值：
  - ENV_FREE：这个 Env 是不活跃的，也就是说在 env_free_list 中。
  - ENV_RUNNABLE：这个 Env 正在等待被处理器运行。
  - ENV_RUNNING：这个 Env 结构体代表了正在运行的环境。
  - ENV_NOT_RUNNABLE：当前环境是活跃的但是不准备运行。比如等待其他环境进行进程间通信。
  - ENV_DYING：Env 是一个僵尸环境，这个环境下一次进入内核的时候会被释放。
- env_pgdir：这个变量存放这个环境的页目录的虚拟地址。

类似于 Unix，一个 jOS 环境中结合了“线程”和“地址空间”的概念。线程是由保护寄存器定义的，而地址空间是由 env_pgdir 指向的页目录和页表定义。

3.1.2 Allocating the Environments Array

在 lab2 中修改了 mem_init() 内为 pages 数组分配了空间，而现在需要进一步修改 mem_init() 来为 Env 分配一个相似的结构 envs。

Exercise 1

修改 kern/pmap.c 中的 mem_init() 来为 envs 分配空间并建立映射。这个数组应该正好包含 NENV 个 Env 结构。并且这个 envs 应该映射到用户制度的 UENVS，这样用户进程可以读取。可以使用 check_kern_pgdir() 来检查代码是否正确。

Exercise 1 实验过程

首先是分配数组，在分配 pages 的代码后添加为 envs 分配空间的代码：

```
1  envs = (struct Env*)boot_alloc(NENV*sizeof(struct Env));
2  memset(envs, 0, NENV*sizeof(struct Env));
```

在分配完内存空间之后，接下来准备映射。因此在对于 pages 的映射之后对 envs 进行映射，添加如下代码：

```
1  boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```

修改完成以后，重新编译运行，结果如图3.1所示。可以看到，显示 check_kern_pgdir() succeeded!，也就是说 exercise1 实验成功。

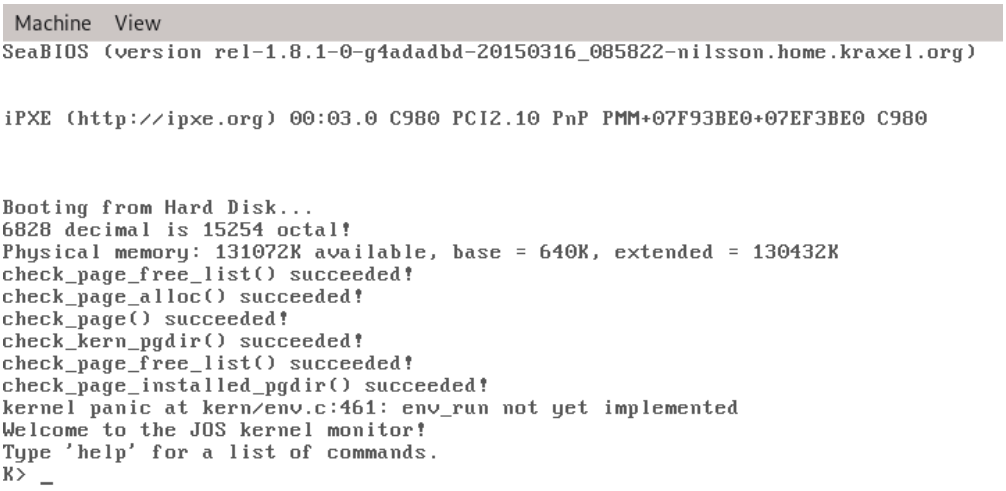

图 3.1: 修改 mem_init 后的运行结果

Exercise 1 实验过程

3.1.3 Creating and Running Environments

现在需要完善 kern/env.c 使之能够运行一个用户环境。由于没有文件系统，因此必须将内核设置为能够加载内核中的静态二进制程序映像文件。

Lab3 里面的 GNUMakefile 文件在 obj/user 目录下生成了一系列二进制文件。通过 kern/Makefrag 能够将这些二进制文件直接链接到可执行文件中。通过链接器中的-b binary 选项能够使这些文件被作为二进制文件链接到内核之后。

Exercise 2

在 env.c 中，完成以下函数：

- env_init()

  初始化所有在 envs 数组中的 Env 结构，并将其加入 env_free_list 中。此外还需要调用 env_init_percput 来配置段式内存管理硬件来将所有的分段分为 0 级（内核）以及 3 级（用户）。

- env_setup_vm()

  分配页目录，初始化用户环境地址空间中和内核相关的部分。

- region_alloc()

  为用户环境分配物理空间。

- load_icode()

  像 boot loader 一样分析一个 ELF 文件，并将它的内容加载到用户环境下。

- env_create()

  使用 env_alloc 和 load_icode 函数分配空间并加载一个 ELF 文件到用户环境中。

- env_run()

  在用户模式下开始一个用户环境。

Exercise 2 实验过程

对于 env_init 函数而言，遍历 envs 数组中的 Env 结构体，把每一个 Env 的 end_id 置 0。实现的 env_init 的代码如下：

```
1  void env_init(void) {
2      int counter;
3      env_free_list = NULL;
4      for (counter = NENV - 1; counter >= 0; --counter) {
5          envs[counter].env_id = 0;
6          envs[counter].env_status = ENV_FREE;
7          envs[counter].env_link = env_free_list;
8          env_free_list = &envs[counter];
9      }
10     // Per-CPU part of the initialization
11     env_init_percpu();
12 }
```

然后填写 env_setup_vm 部分。env_setup_vm 的函数的作用是初始化新的用户环境页目录表，但是只设置夜幕里表中和内核相关的页目录，而不映射用户目录。因此可以使用 kern_pgdir 来设置 env_pgdir 中的内容。最终补充的代码如下：

```
1  ++p->pp_ref;
2  e->env_pgdir = (pde_t *)page2kva(p);
3  memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
```

接下来补充为用户环境分配 len 字节的空间的函数，然后映射到环境中的虚拟地址 va，根据提示，va 向下对齐，va+len 向上对齐。

```
1  static void region_alloc(struct Env *e, void *va, size_t len) {
2      struct PageInfo *page = NULL;
3      va = ROUNDDOWN(va, PGSIZE);
4      void *end = (void *)ROUNDUP(va + len, PGSIZE);
5      for (; va < end; va += PGSIZE) {
6          if (!(page = page_alloc(ALLOC_ZERO)))
7              panic("region_alloc: alloc failed.");
8          if (page_insert(e->env_pgdir, page, va, PTE_U | PTE_W))
9              panic("region_alloc: page mapping failed.");
10     }
11 }
```

load_icode 需要加载 ELF 二进制到用户内存。参考 boot/main.c 中的 boot loader 加载内核到内存，首先验证 ELF 文件的合法性，然后加载 ph-> p_type = ELF_PROG_LOAD 的字段，在加载前需要注意使用 lcr3 切换到用户态的页目录，否则不能够正确的加载到用户内存空间。在加载完成并将多余位清零后，映射初始栈的一个页，最终完成的代码如下：

```
1  static void load_icode(struct Env *e, uint8_t *binary) {
2      struct Elf *elf_header = (struct Elf *)binary;
3      if (elf_header->e_magic != ELF_MAGIC)
4          panic("load_icode: illegal ELF format.");
5      lcr3(PADDR(e->env_pgdir));
6      struct Proghdr *ph = (struct Proghdr *)((uint8_t *)(elf_header) +
           elf_header->e_phoff);
7      struct Proghdr *eph = ph + elf_header->e_phnum;
8      for (; ph < eph; ++ph) {
9          if (ph->p_type == ELF_PROG_LOAD) {
10             region_alloc(e, (void *)ph->p_va, ph->p_memsz);
11             memmove((void *)ph->p_pa, binary + ph->p_offset, ph->p_filesz);
12             memset((void *)(ph->p_pa + ph->p_filesz), 0, ph->p_memsz - ph->
                   p_filesz);
13         }
14     }
15     e->env_tf.tf_eip = elf_header->e_entry;
16     lcr3(PADDR(kern_pgdir));
17     region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
18 }
```

对于 env_create 首先使用 env_alloc 创建一个 env，然后调用 load_icode 来加载 elf 二进制镜像，最后设置 env_type。值得注意的是 env 的父 id 应该设置为 0，其实现如下：

```
1  void env_create(uint8_t *binary, enum EnvType type) {
2      struct Env *environment;
3      if(env_alloc(&environment, 0))
4          panic("env_create: env_alloc failed.");
5      load_icode(environment, binary);
6      environment->env_type = type;
7  }
```

对于 env_run 而言，首先切换判断当前环境是否为空，环境状态是否为 ENV_RUNNING，如果是则将环境设置为 ENV_RUNNABLE，然后将 curenv 设置为当前环境。设置状态 ENV_RUNNING，更新 env_runs 计数器后切换到它的地址空间。使用 env_pop_tf 换源环境寄存器然后进入用户模式。实现的代码如下：

```
1  void env_run(struct Env *e) {
2      if(curenv && curenv->env_status == ENV_RUNNING)
3          curenv->env_status = ENV_RUNNABLE;
4      curenv = e;
5      curenv->env_status = ENV_RUNNING;
```

```
6       ++curenv->env_runs;
7       lcr3(PADDR(curenv->env_pgdir));
8       env_pop_tf(&curenv->env_tf);
9   }
```

Exercise 2 实验过程

用户环境的代码被调用前，操作系统一共按顺序执行了以下几个函数：

- start (kern/entry.S)
- i386_init (kern/init.c)
  - cons_init
  - mem_init
  - env_init
  - trap_init (still incomplete at this point)
  - env_create
  - env_run
    - * env_pop_tf

完成上述函数的代码后重新编译运行，系统会进入用户空间并且开始执行 hello 程序，直到系统调用 int 指令。这个指令不能成功执行，因为 jOS 还没有设置相关硬件来实现从用户态向内核态转换的功能。当 CPU 发现它不能处理这种中断时会触发一个异常，然后发现这个异常也无法处理，直到产生第三个异常，但仍旧不能解决，因此将其叫做"triple fault"。我们可以使用调试器检查我们是否进入了用户模式。使用 make qemu-gdb 并在 env_pop_tf 处设置一个断点，然后单步执行，处理器会在执行完 iret 指令以后进入用户模式。该进入用户模式的第一条指令是一个 cmp 指令。然后使用 b *0x... 设置一个在 obj/user/hello.asm 中的断点中的 sys_cputs 函数的 int $0x30 处。这个 int 指令是一个系统调用，用来向控制台输出一个字符。如果你的程序不能运行到 int 指令说明程序有错误。

按照上述过程进行调试，程序停止在了 int $0x30 处，如图3.2所示。说明程序功能正常。

```
1 ─> 0x800a33:    int    $0x30
2    0x800a35:    pop    %ebx
3    0x800a36:    pop    %esi
4    0x800a37:    pop    %edi
5    0x800a38:    pop    %ebp
6    0x800a39:    ret
**    0x800a33: int    $0x30 (800a33 - 800b32
```

```
Program received signal SIGTRAP, Trace/breakp
oint trap.
The target architecture is assumed to be i386
⇒ 0x800a33:    int    $0x30
0x00800a33 in ?? ()
(gdb)
```

图 3.2: 程序停止在 int $0x30 处

3.1.4 Handling Interrupts and Exceptions

现在需要一个异常处理及系统调用处理机制来使系统从用户态切换到内核态。

Exercise 3

阅读 *Chapter 9, Exceptions and Interrupts* [a]

[a] https://pdos.csail.mit.edu/6.828/2017/readings/i386/c09.htm

3.1.5 Basics of Protected Control Transfer

异常和中断都是保护控制转移，让处理器从用户模式切换到内核模式。这样用户代码不会对内核造成任何影响。在 intel 处理器中，中断通常是外部设备引起的异步的保护控制转移，而异常则是由当前运行的代码引起的同步的保护控制转移。

为了能够保证这些控制转移真的能被保护，处理器的中断/异常机制通常为用户态代码无权选择内核代码的执行起点，处理器只有在某些条件下才能进入内核态。在 x86 上，有 2 中机制配合来提供这种保护：

1. 中断向量表：

   处理器保证撞断和异常只能导致内核进入一些预先定好的入口。x86 处理器可以有最多 256 个不同的中断和异常，而每一个都对应一个唯一的中断向量。一个中断向量的值是根据中断的来源决定的。CPU 将使用这个向量作为中断向量表的索引，而这个表又是内核设置的。通过表项处理器会加载：

   - 加载到 EIP 寄存器的值，也就是指向处理这种类型异常的内核代码指针。
   - 加载到 CS 寄存器的值，包含特权级别 0~1。

2. 任务状态段：

   处理器需要存放中断异常发生之前的旧的处理器状态，包括原 EIP 和 CS 值，以在中断处理之后能够还原到之前的状态。保存的这个位置必须要受到保护，不能随意被修改。

   因此，处理西在处理中断时会导致特权级别由用户转级为内核级，将堆切换到内核内存中。处理器将 SS, ESP, EFLAGS, CS, EIP 和可选的错误码压入堆栈中，然后从中断描述符中加载 CS 和 EIP，设置 ESP 和 SS 指向新的堆栈。

3.1.6 Types of Exceptions and Interrupts

所有的 x86 处理器内部产生的议程向量是 0~31 之间的整数，也映射到了 IDT 的 0~31 项。大于 31 的项只被软件中断所使用，也就是说可以被 int 触发，或者是异步的硬件中断。

这一节将要扩展 jOS 的功能使之能够处理 0~31 号的内部异常。下一节会让 jOS 处理 48 号软件中断。

3.1.7 An Example

在这个例子中，假设处理器遇到了除 0 的问题。

1. 处理器切换到 TSS 的 SS0 和 ESP0 对应的堆栈，在 jOS 中，这两个字段是 GD_KG 和 KSTACKTOP。

2. 处理器将异常参数压入内核堆栈，并放在 KSTACKTOP 中。

```
    +--------------------+ KSTACKTOP
    | 0x00000 | old SS   |     " - 4
    |      old ESP       |     " - 8
    |     old EFLAGS     |     " - 12
    | 0x00000 | old CS   |     " - 16
    |      old EIP       |     " - 20 <---- ESP
    +--------------------+
```

3. 由于处理器错误在 x86 上是 0 号中断向量，因此去读 IDT 的第 0 项并设置 CS:EIP 指向中断处理程序。

4. 处理函数接过控制权并处理异常。

对于确定类型的 x86 异常，除了上面标准的 5 个压栈元素外还有一个错误码。当处理器将错误码压栈时，栈是这样的：

```
    +--------------------+ KSTACKTOP
    | 0x00000 | old SS   |     " - 4
    |      old ESP       |     " - 8
    |     old EFLAGS     |     " - 12
    | 0x00000 | old CS   |     " - 16
    |      old EIP       |     " - 20
    |     error code     |     " - 24 <---- ESP
    +--------------------+
```

3.1.8 Nested Exceptions and Interrupts

处理器在内核模式和用户模式都可以处理异常和中断。但是当内核从用户态进入内核态的时候，x86 处理器会在压入旧的寄存器之前自动切换栈并通过 IDT 触发异常处理。如果当中断或异常发生时处理器已经在内核态了，那么 CPU 会在同一个栈上压入更多的值。这样，内核就能够处理嵌套中断；了。如果处理器已经在内核模式且正在处理嵌套异常，就不会保存 SS 和 ESP 寄存器，因此堆栈如下：

```
    +--------------------+ <---- old ESP
    |     old EFLAGS     |     " - 4
    | 0x00000 | old CS   |     " - 8
    |      old EIP       |     " - 12
    +--------------------+
```

如果处理器在内核模式处理异常，但栈空间不足，不能将旧的状态压入堆栈，那么处理器之后就不能恢复，只能重启。内核应该被设计为不允许这种事情发生。

3.1.9 Setting Up the IDT

现在可以设置 IDT 表并处理 JOS 的内部异常了（中断向量 0~31）。最终需要实现的代码效果如下：

```
        IDT              trapentry.S       trap.c
    +----------------+
    |   &handler1    |---> handler1:        trap (struct Trapframe *tf)
    |                |       // do stuff    {
    |                |       call trap        // handle the exception/interruput
    |                |       // ...         }
    +----------------+
    |   &handler2    |---> handler2:
    |                |      // do stuff
    |                |      call trap
    |                |      // ...
    +----------------+
           ...
    +----------------+
    |   &handlerX    |---> handlerX:
    |                |       // do stuff
    |                |       call trap
    |                |       // ...
    +----------------+
```

每一个中断或者异常结构都有它的中断处理函数，定义在 trapentry.S 中。trap_init() 初始化 IDT 表。每个处理函数都应该构建一个在 Trapframe 堆栈上的结构体，并调用 trap() 函数指向它。trap() 则处理异常/中断。

Exercise 4

编辑 trap.S 以及 trap.c 并实现上述功能。对于每一个定义在 inc/trap.h 中的 trap，都应该有一个函数入口应该被加在 trapentry.S 中。应该提供一个 _alltraps 供 TRAPHANDLER 宏引用。要初始化 idt 表需要修改 trap_init 函数，使表中的每一项指向定义在 trapentry.S 中的入口指针。实现的 _alltraps 函数应该：

1. 将值压入堆栈，使堆栈看起来像一个 Trapframe
2. 加载 GD_KD 进入%ds 以及%es
3. 使用 pusl %esp 给 Trapframe 传递指针，作为 trap() 的参数
4. 调用 trap

Exercise 4 实验过程

首先，trapentry.S 中的宏定义 TRAPHANDLER 以及 TRAPHANDLER_NOEC 定义了发生中断或异常时用于初始处理的函数。因为有些中断有错误码，有些没有，因此需要两个函数。通过参考 *80386*

Programmer' s Manual 9.10 Error Code Summary[1]可以知道哪些中断有错误码。因此 trapentry.S 修改如下：

```
1  TRAPHANDLER_NOEC(handler_divide,  T_DIVIDE)
2  TRAPHANDLER_NOEC(handler_debug,   T_DEBUG)
3  TRAPHANDLER_NOEC(handler_nmi,     T_NMI)
4  TRAPHANDLER_NOEC(handler_brkpt,   T_BRKPT)
5  TRAPHANDLER_NOEC(handler_oflow,   T_OFLOW)
6  TRAPHANDLER_NOEC(handler_bound,   T_BOUND)
7  TRAPHANDLER_NOEC(handler_illop,   T_ILLOP)
8  TRAPHANDLER_NOEC(handler_device,  T_DEVICE)
9  TRAPHANDLER_NOEC(handler_simderr, T_SIMDERR)
10 TRAPHANDLER_NOEC(handler_fperr,   T_FPERR)
11 TRAPHANDLER_NOEC(handler_mchk,    T_MCHK)
12 TRAPHANDLER_NOEC(handler_syscall, T_SYSCALL)
13 TRAPHANDLER(handler_dblflt, T_DBLFLT)
14 TRAPHANDLER(handler_tss,    T_TSS)
15 TRAPHANDLER(handler_segnp,  T_SEGNP)
16 TRAPHANDLER(handler_stack,  T_STACK)
17 TRAPHANDLER(handler_gpflt,  T_GPFLT)
18 TRAPHANDLER(handler_pgflt,  T_PGFLT)
19 TRAPHANDLER(handler_align,  T_ALIGN)
20
21 _alltraps:
22     pushl %ds
23     pushl %es
24     pushal
25     movw $GD_KD, %eax
26     movw %ax, %ds
27     movw %ax, %es
28     pushl %esp
29     call trap
```

然后在 trap.c 中完成 trap_init，对于系统的 IDT 表进行初始化：

```
1  void handler_divide();
2  void handler_debug();
3  void handler_nmi();
4  void handler_brkpt();
5  void handler_oflow();
6  void handler_bound();
7  void handler_illop();
8  void handler_device();
9  void handler_simderr();
10 void handler_fperr();
11 void handler_mchk();
```

[1] https://pdos.csail.mit.edu/6.828/2016/readings/i386/s09_10.htm

```
12 void handler_syscall();
13 void handler_dblflt();
14 void handler_tss();
15 void handler_segnp();
16 void handler_stack();
17 void handler_gpflt();
18 void handler_pgflt();
19 void handler_align();
20
21 void
22 trap_init(void)
23 {
24     extern struct Segdesc gdt[];
25
26     SETGATE(idt[T_DIVIDE],  0, GD_KT, handler_divide,  0);
27     SETGATE(idt[T_DEBUG],   0, GD_KT, handler_debug,   0);
28     SETGATE(idt[T_NMI],     0, GD_KT, handler_nmi,     0);
29     SETGATE(idt[T_BRKPT],   0, GD_KT, handler_brkpt,   3);
30     SETGATE(idt[T_OFLOW],   0, GD_KT, handler_oflow,   0);
31     SETGATE(idt[T_BOUND],   0, GD_KT, handler_bound,   0);
32     SETGATE(idt[T_ILLOP],   0, GD_KT, handler_illop,   0);
33     SETGATE(idt[T_DEVICE],  0, GD_KT, handler_device,  0);
34     SETGATE(idt[T_SIMDERR], 0, GD_KT, handler_simderr, 0);
35     SETGATE(idt[T_FPERR],   0, GD_KT, handler_fperr,   0);
36     SETGATE(idt[T_MCHK],    0, GD_KT, handler_mchk,    0);
37     SETGATE(idt[T_SYSCALL], 0, GD_KT, handler_syscall, 3);
38     SETGATE(idt[T_DBLFLT],  0, GD_KT, handler_dblflt,  0);
39     SETGATE(idt[T_TSS],     0, GD_KT, handler_tss,     0);
40     SETGATE(idt[T_SEGNP],   0, GD_KT, handler_segnp,   0);
41     SETGATE(idt[T_STACK],   0, GD_KT, handler_stack,   0);
42     SETGATE(idt[T_GPFLT],   0, GD_KT, handler_gpflt,   0);
43     SETGATE(idt[T_PGFLT],   0, GD_KT, handler_pgflt,   0);
44     SETGATE(idt[T_ALIGN],   0, GD_KT, handler_align,   0);
45
46     // Per-CPU setup
47     trap_init_percpu();
48 }
```

完成后使用 make grade 进行测试, 可以发现 divzero, softint 以及 badsegment 测试通过, 如图3.3所示, 说明 IDT 初始化正确。

```
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/husixu/Homework/jOS/lab3'
divzero: OK (2.4s)
    (Old jos.out.divzero failure log removed)
softint: OK (1.4s)
    (Old jos.out.softint failure log removed)
badsegment: OK (1.3s)
    (Old jos.out.badsegment failure log removed)
Part A score: 30/30
```

图 3.3: 使用 make grade 进行测试

Exercise 4 实验过程

Question

1. 对于每一个中断/异常都设置一个独立的处理函数的意义是什么?
2. 你做了让 user/softint 正确执行的工作吗? grade script 希望它产生一个 general protection falt(trap 13)，但是 softint 中为 int $14。为什么产生了中断向量 13? 如果系统允许 int $14 调用 kernel page fault 处理函数?

Answer

1. 因为不同的中断可能需要不同的处理方式，比如有些中断需要返回，有些中断则不需要，还有些中断需要做额外的工作。
2. 应为当先系统在用户态，而 int 为特权级别为 0 的指令，此时不能直接调用 int 指令，会引发 general protection exception。如果允许 int $14 处理，那么会导致用户态程序可能得到 0 级特权，造成保护失效。

Answer

3.2 Page Faults, Breakpoints Exceptions, and System Calls

3.2.1 Handling Page Faults

当处理器产生一个缺页异常时，它会将因此缺页异常的线性地址（虚拟的）存入 CR2 中。在 trap.c 中我们提供了一个特殊的函数 page_fault_handler() 用于处理缺页异常。

Exercise 5

修改 trap_dispatch 函数使系统能够把缺页异常分发到 page_fault_handler 上。修改完成后运行 make grade 应该可以成功通过 faultread, faultreadkernel, faultwrite, 以及 faultwritekernel 检查。

Exercise 5 实验过程

trap_dispatch 是一个分发函数，通过 Trapframe 指针 tf 中的 tf_trapno 来判断这个中断是什么中断。而在这一个 exercise 中，如果中断是缺页中断则调用 page_fault_handler 函数。考虑到之后可能要添加的中断类型，在 trap_dispatch 中添加的代码如下：

```
switch(tf->tf_trapno){
    case T_PGFLT:
        page_fault_handler(tf);
        break;
}
```

重新编译运行后，通过了题目描述中的 4 个测试，如图3.4所示。说明这部分代码实现成功。

```
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/husixu/Homework/jOS/lab3'
divzero: OK (1.2s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.1s)
    (Old jos.out.faultread failure log removed)
faultreadkernel: OK (1.0s)
    (Old jos.out.faultreadkernel failure log removed)
faultwrite: OK (1.1s)
    (Old jos.out.faultwrite failure log removed)
faultwritekernel: OK (0.8s)
    (Old jos.out.faultwritekernel failure log removed)
breakpoint: FAIL (1.1s)
    AssertionError: ...
```

图 3.4: 重新编译运行后的 make grade 输出

Exercise 5 实验过程

3.2.2 The Breakpoint Exception

异常编号为 3 的断点异常能够让调试器给程序加上断点，也就是将要加断点的语句用一个 int3 指令替换，然后在执行到 int3 时触发中断。在 jOS 中需要将这个中断变为任何用户环境都能调用的伪系统调用。

Exercise 6

修改 trap_dispatch，使断点异常发生时能够调用 kernel monitor。修改完成后重新 make grade 应该能够通过 breakpoint 测试。

Exercise 6 实验过程

与 exercise 5 类似，但是这里处理的是 T_BRKPT。调用 kernel monitor 需要使用 kern/monitor.c 中的 monitor 函数。修改后的 trap_dispatch 中的前半部分内容如下：

```
switch(tf->tf_trapno){
    case T_PGFLT:
        page_fault_handler(tf);
        break;
    case T_BRKPT:
        monitor(tf);
        break;
}
```

修改完成后，成功通过 breakpoint 测试，如图3.5所示。

```
make[1]: Leaving directory '/home/husixu/Homework/jOS/lab3'
divzero: OK (1.4s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.0s)
faultwrite: OK (0.8s)
faultwritekernel: OK (1.1s)
breakpoint: OK (1.0s)
```

图 3.5: 程序通过 breakpoint 测试输出

Exercise 6 实验过程

Question

3. breakpoint exeception 测试用例会征程一个 breakpoint 异常或者 general protection 错误，依赖于如何初始化 IDT 中的 breakpoint entry。为什么？要怎么做才能让 breakpoint exception 正常工作？怎样的错误设置会导致触发 general protection error?
4. 这些机制有什么意义？尤其是对于 user/softint 中的测试程序而言？

Answer

3. 如果在 IDT 中设置 breakpoint exception 时将 DPL 字段设置为 0 则会触发 breakpoint exception，设置为 3 则会触发 general protection exception。DPL 字段为段描述符优先级，如果当前程序为用户态但是尝试调用内核态的指令的时候就会触发 general protection exception。只有当前程序的优先级小于或等于段描述符优先级才能触发正确的 breakpoint exception。
4. 这些机制保证用户环境不能随意访问内核态的代码和内存，保护内核不受用户程序的破坏。

Answer

3.2.3 System calls

用户程序会要求内核通过系统调用的方式帮其完成一些任务。当用户程序触发系统调用时，处理器进入内核态并保存用户的处理状态。内核处理完成后返回用户程序，但具体的细节随系统的不同而不同。

jOS 使用 int 来处理系统调用。特别的，使用 int $0x30 作为系统调用中断。常量 T_SYSCALL 就是 0x30。0x30 不能被外部硬件产生，因此没有任何歧义。

应用程序会把系统调用和参数放到寄存器中，通过这种方法内核就不需要查询用户程序的堆栈了。系统调用号存放到%eax 中，参数则存放在%edx, %ecx, %ebx, %edi, 和%esi 中。返回值存放到%eax@s 中。lib/syscall.c 中已有了触发系统调用的方法。

Exercise 7

通过编辑 kern/trapentry.S 以及 kern/trap.c 的 trap_init()，给 T_SYSCALL 添加一个中断向量处理函数。同时 trap_dispatch 也需要被修改，通过调用 syscall 的方法来处理系统调用。最后，需要在 kern/syscall.c 中首先实现 syscall 函数。如果系统调用号不合法，需要 syscall 返回-E_INVAL。

通过 make run-hello 运行 user/hello，qemu 应该打印处 hello, world，并触发一个 page fault。并且 make grade 应该能够通过 testbss 测试。

Exercise 7 实验过程

在用户态执行系统调用时，首先产生了中断 30，因此在 kern/trapentry.S 中添加一个处理函数声明 TRAPHANDLER_NOEC(handler_syscall, T_SYSCALL)，并在 trap_init 中添加 handler_syscall 的声明以及在 trap_init 中通过 SETGATE(idt[T_SYSCALL], 0, GD_KT, t_syscall, 3); 将其注册到 IDT，这些在 exercise 5 中已经完成，此时系统已经能够正确捕捉 int 30 了。

观察 lib/syscall.c 的 syscall，发现其就是执行了 int 指令并取回了返回值，而对于这条 int 指令的处理，则是在 kern/syscall.c 中进行的。首先，完成 kern/trap.c 中对于中断的分发，也就是在 switch 中加入如下几行：

```
1 case T_SYSCALL:
2     tf->tf_regs.reg_eax = syscall(
3             tf->tf_regs.reg_eax,
4             tf->tf_regs.reg_edx,
5             tf->tf_regs.reg_ecx,
6             tf->tf_regs.reg_ebx,
7             tf->tf_regs.reg_edi,
8             tf->tf_regs.reg_esi);
9     break;
```

然后在 kern/syscall.c 中完成对于 syscall 的实现，从而完成对于整个 int 指令的调用：

```
1 int32_t syscall(uint32_t syscallno, uint32_t a1,
2         uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5) {
3     switch (syscallno) {
4         case SYS_cputs:
5             sys_cputs((char *)a1, a2);
```

```
6               return 0;
7           case SYS_cgetc:
8               return sys_cgetc();
9           case SYS_getenvid:
10              return sys_getenvid();
11          case SYS_env_destroy:
12              return sys_env_destroy(a1);
13          default:
14              return -E_INVAL;
15      }
16  }
```

填写完成后，重新编译并运行 qemu，输出 hello world 并触发缺页中断。如图3.6所示。

```
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffc
hello, world
Incoming TRAP frame at 0xeffffffc
[00001000] user fault va 00000048 ip 0080004b
TRAP frame at 0xf0223000
```

图 3.6: 输出 hello world 并触发缺页中断

运行 make grade，成功通过 testbss 测试，如图3.7所示，说明系统调用实现成功。

```
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.1s)
faultwrite: OK (0.8s)
faultwritekernel: OK (1.1s)
breakpoint: OK (1.0s)
testbss: OK (1.0s)
hello: FAIL (1.2s)
    AssertionError: ...
         check_page_free_list() succeeded!
```

图 3.7: 成功通过 testbss 测试

Exercise 7 实验过程

3.2.4 User-mode startup

用户模式开始运行的地方是 lib/entry.S。在该文件中首先进行一些设置，然后调用 libmain。应该修改 libmain() 来初始化全局指针 thisenvz 指向 envs 数组中的 Env 结构体。

然后 libmain 调用 main，也就是 user/hello.c 中被调用的函数。在之前的实验中发现 hello.c 只能打印 hello world 并报出 page fault 异常，而其原因就是 this->env_id 语句。如果正确初始化了 thisenv 就不会报错了。

Exercise 8

补全用户库中的代码并启动内核，user/hello 应该打印出 hello, world 然后打印 i am environment 00001000。通过调用 sys_env_destroy()，user/hello 会尝试退出。由于内核仅仅支持一个用户环境，因此它应该显示用户环境已被销毁的信息，然后退回 kernel monitor。完成后 make grade 应该能够通过 hello test 测试。

Exercise 8 实验过程

在 libmain 中，修 thisenv 让其指向 env 当前环境的 env 即可。使用 sys_getenvid 来获得当前的环境 id。因此将 libmain 中的 thisenv 修改为如下即可。

```
1  thisenv = envs + ENVX(sys_getenvid());
```

重新编译运行，并运行 make grade，输出如图3.8所示，此时能够通过 hello 测试。

```
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.0s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.0s)
breakpoint: OK (1.0s)
testbss: OK (1.0s)
hello: OK (1.0s)
buggyhello: FAIL (1.0s)
    AssertionError: ...
```

图 3.8: 程序通过 hello 测试

Exercise 8 实验过程

3.2.5 Page faults and memory protection

内存保护是操作系统的一个非常重要的特性，能够保证 bug 不能够损坏其他的程序或者操作系统。操作系统通常依赖于硬件来完成这一功能。操作系统能够让硬件知道那笑虚拟地址是有效的。当程序尝试访问一个无效地址或者越权操作时就会触发异常。如果异常是可以修复的，那么就会修复异常并继续运行程序，否则就不会继续运行。

在许多操作系统中，内核在初始情况下只会分配一个内核堆栈。如果程序想要访问这个堆栈之外的堆栈空间，就会触发异常，内核会自动分配页个程序然后继续让程序运行。

但是系统调用需要存在问题，大部分系统系统调用接口让用户传递一个指向用户缓冲区的指针给内核，但是：

1. 内核中的 page fault 比用户中的 page fault 严重。如果内核出现 page fault，那么这是内核 bug，而且异常处理会中断内核的执行。但是当内核解引用用户程序的时候，它需要一种方法标记这些 page fault 确实是由用户引起的。

2. 内核通常比用户程序由更高级别的权限。用户程序可能会传递一个内核可读写但是用户不行的指针。此时内核不能对其进行解引用，否则可能泄露内核信息。

Exercise 9

修改 kern/trap.c，使其能够检测内核模式下的 page fault 发生并发出 kernel panic。阅读 kern/pmap.c 中的 use_mem_assert 并实现其中的 use_mem_check。修改 ker/syscall.c 检查输入参数。

启动内核并运行 user/buggyhello。环境应该被摧毁，但是不应该出现 kernel panic。应该能够看到：

```
[00001000] user_mem_check assertion failure for va00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

最后，修改 kern/kdebug.c 中的 debuginfo_eip 来运行 user_mem_check 检查 use, stabs 和 stabstr。如果现在运行 user/breakpoint，应该能够从内核监视器中运行 backtrace 来检查在 page fault 之前检查 lib/libmain.c。是什么导致了 page fault?

Exercise 9 实验过程

首先检查 page fault 是否在内核模式，即检查 Trap Frame 中的 tf_cs，在 page_fault_handler 中添加的代码如下：

```
1  if(tf->tf_cs == GD_KT)
2      panic("page_fault_handler: kernel page fault");
```

pmap.c 中的 user_mem_check。阅读 user_mem_assert, 可以发现它调用了 user_mem_check。然而 user_mem_check 的功能当前永辉态程序是否有对于 $[va, va+len)$ 的 perm|PTE_P 的访问权限。因此在 user_mem_check 中查看用户态程序中的页表项，然后检查其 perm|PTE_P。最终实现的程序如下：

```
1  int user_mem_check(struct Env *env, const void *va, size_t len, int perm) {
2      uint32_t start = (uint32_t)ROUNDDOWN(va, PGSIZE);
3      uint32_t end = (uint32_t)ROUNDUP(va + len, PGSIZE);
4      pte_t *page;
5      for (; start < end; start += PGSIZE) {
6          page = pgdir_walk(env->env_pgdir, (void *)start, 0);
7          if (!page || start > ULIM || ((uint32_t)(*page) & perm) != perm ) {
8              if (start <= (uint32_t)va)
9                  user_mem_check_addr = (uintptr_t)va;
10             else
11                 user_mem_check_addr = (uintptr_t)start;
12             return -E_FAULT;
13         }
14     }
15     return 0;
16 }
```

接下来对于 kern/syscall.c 进行补全。通过观察发现需要补全的是 sys_cputs 函数。通过注释可以发现需要用户程序检查用户对于虚拟地址空间 $[s, s+len)$ 是否具有访问权限，而这个则可以用上面实现的 user_mem_assert 实现。这个函数补全后如下：

```
1  static void sys_cputs(const char *s, size_t len) {
2      user_mem_assert(curenv, s, len, 0);
3      cprintf("%.*s", len, s);
4  }
```

最后修改 kern/kdegbug.c 中的 debuginfo_eip。添加如下代码：

```
1  if(user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U))
2      return -1;
```

运行 make run-breakpoint，显示如图3.9所示。可以看到，输入 backtrace 能够显示 backtrace 之前能够追踪进入 libmain.c

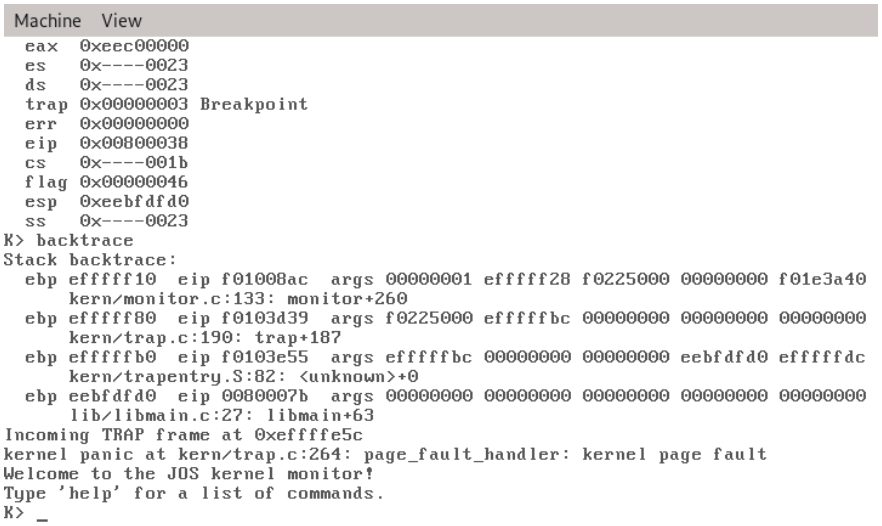

图 3.9: 运行 user/breakpoint 后的结果

通过 gdb 进行追踪，发现是由于执行 mon_backtrace 时进行追踪时到达了用户栈顶，然后在打印参数时访问的第六个参数超过了用户栈的大小，如图3.10所示。

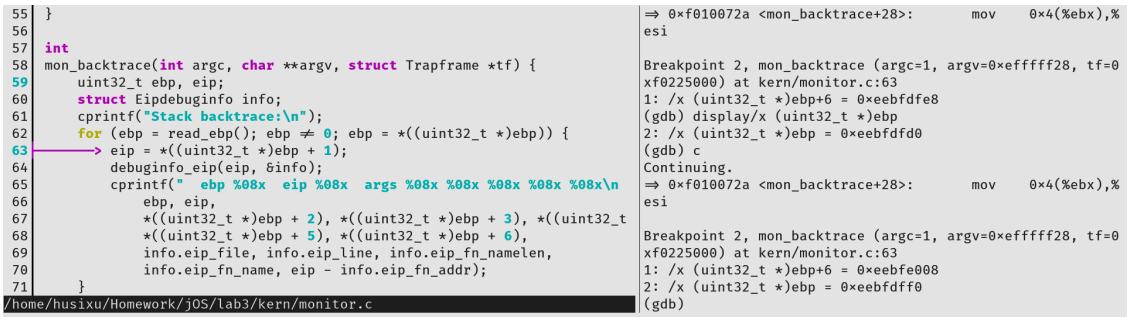

图 3.10: 使用 gdb 进行追踪

最后，使用 make grade 进行测试，发现能够正常通过所有测试，如图3.11所示。

```
divzero: OK (1.1s)
softint: OK (1.0s)
badsegment: OK (0.8s)
Part A score: 30/30

faultread: OK (1.1s)
faultreadkernel: OK (1.0s)
faultwrite: OK (1.1s)
faultwritekernel: OK (0.8s)
breakpoint: OK (1.2s)
testbss: OK (0.9s)
hello: OK (1.1s)
buggyhello: OK (1.0s)
buggyhello2: OK (1.0s)
evilhello: OK (1.0s)
Part B score: 50/50

Score: 80/80
```

图 3.11: make grade 通过所有测试

Exercise 9 实验过程

Exercise 10

启动你的内核并运行 user/evilhello。环境应该被摧毁并且内核不应该 panic。你应该能够看到：

```
[00000000] new env 00001000
...
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
```

Exercise 10 实验过程

运行 evilhello，结果如图3.12所示。环境被摧毁且没有发生 kernel panic。

```
Machine  View
SeaBIOS (version rel-1.8.1-0-g4adadbd-20150316_085822-nilsson.home.kraxel.org)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F93BE0+07EF3BE0 C980


Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffc
Incoming TRAP frame at 0xeffffffc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 3.12: evilhello 运行结果

Exercise 10 实验过程

第 4 章　Preemptive Multitasking

4.1　Multiprocessor Support and Cooperative Multitasking

这一部分首先要向 jOS 扩展到一个多处理器的系统上，然后实现一些内核调用来允许用户级别的环境来创建新的环境。还需要实现轮训调度，在进程不使用 CPU 使允许内核切换到另一个进程。

4.1.1　Multiprocessor Support

我们将使 jOS 支持对称多处理 (SMP) 系统。这种系统能够使所有的 CPU 对内存能和 I/O 够有相同的访问权限。虽然 CPU 在对称多处理的情况下以相同的方式工作，但是在启动过程中可以被分为 2 个类型：引导处理器 (BSP) 以及应用处理器 (AP)。引导处理器用户初始化以及启动操作系统，而应用处理器被引导处理器启动。哪个处理器作为引导处理器由硬件决定。

在对称多处理系统中，每个 CPU 都有一个本地 APIC(LAIPC) 单元。LAIPC 单元用于传递中断并给它连接的 CPU 一个唯一的 id。在这个 lab 中将使用 LAIPC 以下功能：

- 从 BSP 读取 LAPIC 标识符来区分当前代码在那个 CPU 上运行。
- 从 BSP 发送 STARTUP 跨处理器中断到 AP 来启动其他 CPU。
- 使用 LAPIC 的内置计时器编程来触发时钟中断以支持抢占式多任务处理。

处理器通过应设在内存上的 I/O(MMIO) 来访问 LAPIC。在 MMIO 中，物理内存的一部分被链接到 I/O 设备的寄存器。因此 load/store 指令可以用于访问设备的寄存器。LAPIC 起始于物理地址 0xfe000000。对于以前的直接到 KERNBASE 的映射来说这太高了。jOS 在虚拟地址 MMIOBASE 留了 4MB 的空间，因此我们由空间来映射这一类的设备。

> **Exercise 1**
>
> 实现 kern/pmap.c 中的 mmio_map_region。查看 kern/lapic.c 来查看如何使用它。在对这一 exercise 进行测试之前需要先完成下一个 exercise。

Exercise 1 实验过程 ——————————————————————————

首先完成 mmio_map_region。调用 boot_map_region 建立需要的映射。根据注释，需要注意使用的权限为 PTE_PCD|PCE_PWT 来防止 CPU 来 cache 这部分内存，此外还需要注意每次都需要更新 base，以及超出内存的部分需要发出 panic。

```
1  void * mmio_map_region(physaddr_t pa, size_t size) {
2      static uintptr_t base = MMIOBASE;
3
4      if(base + ROUNDUP(size, PGSIZE) > MMIOLIM)
5          panic("mmio_map_region: mmio overflow.");
6      boot_map_region(kern_pgdir, base, ROUNDUP(size, PGSIZE), pa, PTE_W|PTE_PCD|
           PTE_PWT);
7      uintptr_t temp = base;
8      base += ROUNDUP(size, PGSIZE);
9      return (void *)temp;
10 }
```

Exercise 1 实验过程

Application Processor Bootstrap

在启动 AP 之前，BSP 理器首先需要收集处理器系统的信息，包括 CPU 总数，CPU 的 APIC id，LAPIC 的 MMIO 地址。kern/mpconfig.c 中的 mp_init() 通过读取 BIOS 中的 MP 配置来获得这些信息。APs 从实模式开始

boot_aps 函数启动了 APs 的引导。AP 的音高过程类似于 boot/boot.S 中的 bootloader 启动过程。因此 boot_aps() 将 AP 的入口复制到实模式可以访问的区域 MPENTRY_PADDR。然后 boot_aps() 通过发送 STARTUP 跨处理器中断到 LAPIC 单元，逐个激活 APs。激活时，首先初始化 AP 的 CS:IP 让其从入口执行代码，然后开启分页进入保护模式，然后调用 mp_main()，最后等待 AP 的 CPU_STARTED 信号，并开始激活下一个。

Exercise 2

阅读 kern/init.c 中的 boot_aps() 以及 mp_main，以及 kern/mpentry.S 中的汇编。然后修改 kern/pmap.c 中的 page_init() 实现，来避免将 MPENTRY_PADDR 加入空闲链表，一次确保能够安全的复制 AP 的启动代码到这个物理地址并运行。代码应该能够通过更新过的 check_page_free_list 测试。(但可能不会通过 check_kern_pgdir 测试)

Exercise 2 实验过程

将从 MPENTRY_PADDR 开始的一个物理页标记为已使用，并能够将其加入链表。而 MPENTRY_PADDR=0x7000，在低地址处，因此将处理低地址的代码的循环中加入一个判断条件即可，即修改为如下代码:

```
1  for (i = 1; i < npages_basemem; ++i) {
2      if(i == MPENTRY_PADDR/PGSIZE){
3          pages[i].pp_ref = 1;
4          continue;
5      }
6      pages[i].pp_ref = 0;
7      pages[i].pp_link = page_free_list;
```

```
8       page_free_list = &pages[i];
9   }
```

修改完成后重新编译并运行 qemu，其显示输出如图4.1所示。

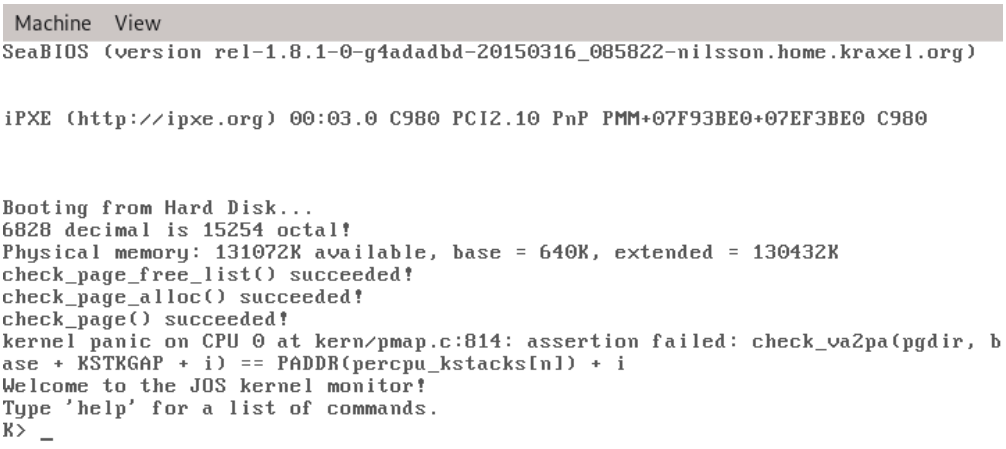

图 4.1: 通过 check_page_free_list 测试

Exercise 2 实验过程

Question

1. 比较 kern/mpentry.S 以及 boot/boot.S。记住 kern/mpentry.S 的目标是向内核一样在 KERNBASE 之上运行。宏 MPBOOTPHYS 的作用是什么？为什么在 kern/mpentry.S 中它是必要的但是 boot/boot.S 中它不是必要的？也就是说，如果 kern/mpentry.S 中不写 MPBOOTPHYS 会发生什么？

Answer

1. 根据 MPBOOTPHYS 的内容可知，起作用是将高地址转换为绝对地址址。因为此时还出于实模式，但是代码中的地址为高地址，因此需要进行转换。

Answer

Per-CPU State and Initialization

在写一个多处理器操作系统时，区分 CPU 的私有状态以及全局状态非常关键。kern/cpu.h 定义了大部分的私有状态。应该注意的私有状态有：

- Per-CPU 内核栈。

  多个 CPU 可能同时陷入内核状态，因此需要给每个处理器一个独立于其他 CPU 的的内核栈。数组 percpu_kstacks[NCPU][KSTKSIZE] 为 NCPU 个内核栈保留了空间。在 Lab2 中 BSP 的内核栈映射到了 KSTACKTOP 下方，而在 lab4 中需要把每个 CPU 的内核栈都映射到这个区域。每个栈之间留下一个空也作为缓冲区避免溢出。

- Per-CPU TSS 和 TSS 描述符

  为了标识每个 CPU 内核栈的位置，需要任务状态段 (TSS)

- Per-CPU 当前环境指针

  每个 CPU 能够同时运行各自的用户进程因此重新定义了 curenv。

- Per-CPU 系统寄存器

  所有的寄存器包括系统寄存器都是 CPU 私有的，因此初始化寄存器的指令需要在每个 CPU 执行一次。

Exercise 3

修改 kern/pmap.c 中的 mem_init_mp() 来映射开始于 KSTACKTOP 的 per-CPU 内核栈。每个栈的大小是 KSTKSIZE 加上 KSTKGAP 字节的不被映射的空间。代码应该能够通过新的 check_kern_pgdir() 测试。

Exercise 3 实验过程

在这一部分代码中，只需要像 lab2 一样逐个映射每一个栈的内存地址即可。在这一部分中，虽然 BSP 的栈地址之前已经映射过，但是更换物理地址不会增删页面引用，因此不修改之前的引用也不会出现问题。

```
1 static void mem_init_mp(void) {
2     uint32_t i;
3     uintptr_t start = KSTACKTOP-KSTKSIZE;
4     for(i=0; i < NCPU; ++i){
5         boot_map_region(kern_pgdir, (uintptr_t)(start), KSTKSIZE, PADDR(
              percpu_kstacks[i]), PTE_W);
6         start -= (KSTKSIZE + KSTKGAP);
7     }
8 }
```

补完这一部分代码后重新编译运行 qemu，结果如图4.2所示，能够成功通过 check_kern_pgdir() 测试。

```
Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:322: page_fault_handler: kernel page fault
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 4.2: 通过 check_kern_pgdir 测试

Exercise 3 实验过程

Exercise 4

kern/trap.c 中的 trap_init_percpu() 代码为了 BSP 初始化 TSS 以及 TSS 描述符。在 Lab3 中它是工作正确的，但是在运行其他 CPU 是运行不正确。修改这部分代码使它能在所有 CPU 上运行。

Exercise 4 实验过程

根据注释中的提示我们可以知道 thiscpu 总是指向当前 cpu 的信息，而第 i 个 CPU 的 TSS 描述符为 $gdt[(GD\_TSS0 >> 3) + i]$，再参考用于初始化 BSP 的代码，修改后的代码如下：

```
1  void trap_init_percpu(void) {
2      struct Taskstate *thists = &thiscpu->cpu_ts;
3
4      thists->ts_esp0 = KSTACKTOP - thiscpu->cpu_id * (KSTKSIZE + KSTKGAP);
5      thists->ts_ss0 = GD_KD;
6      thists->ts_iomb = sizeof(struct Taskstate);
7
8      gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id] =
9          SEG16(STS_T32A, (uint32_t) (thists), sizeof(struct Taskstate) - 1, 0);
10     gdt[(GD_TSS0 >> 3) + thiscpu->cpu_id].sd_s = 0;
11
12     ltr(GD_TSS0 + (thiscpu->cpu_id << 3));
13     lidt(&idt_pd);
14 }
```

修改完成后按照指导输入 make qemu CPUS=4，输出结果如图4.3所示。说明

```
Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:323: page_fault_handler: kernel page fault
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

图 4.3: 使用 CPUS=4 启动 qemu 界面

Locking

当前代码在初始化 AP 之后就会开始自旋，在让 AP 进一步执行之前，首先需要解决多个 CPU 运行内核的地址竞态。最简单的方法是使用 big kernel lock，也就是使用一个全局的锁来锁住整个内核，并在环境返回用户模式时释放这个锁。此时用户环境可以运行在多个 CPU 上但是内核只能运行在一个 CPU 上。

kern/spinlock.h 声明了这个 big kernel lock，名称为 kernel_lock。它也提供了 lock_kernel 以及 unlock_kernel 来加锁和解锁。在以下四个地方应该加锁：

- i386_init() 中，BSP 唤醒其他 CPU 之前。
- mp_main() 中，在初始化 AP 之后加锁，并调用 sched_yield 来在这个 AP 上运行用户环境。
- 在 trap() 中，从用户模式进入内核模式需要加锁。检查 tf_cs 的较低比特来判断当前是在用户环境中还是在内核中。
- 在 env_run() 中，在切换到用户模式前解锁。不要太晚或太早解锁，否则可能产生死锁或竞态。

Exercise 5

按照上述描述使用 big kernel lock。使用 lock_kernel() 以及 unlock_kernel() 在适当的地方加锁和解锁。

Exercise 5 实验过程

按照提示在这一节所描述的 3 个地方（i386_init, mp_main 以及 trap 中）加锁即可。注意的是 unlock_kernel() 应该置于 env_pop_tf(&curenv->env_tf); 之前而不是其他位置。重新编译并运行代码可以发现 exercise4 中出现的 kernel panic 消失了，但也由于锁的原因并没有进入 user mode。如图4.4所示。

```
Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
```

图 4.4: 加上 big kernel lock 后运行 qemu 结果

Exercise 5 实验过程

Question

2. 看来使用大内核锁能够保证一次只有一个 CPU 能够运行内核代码。为什么我们还是需要为每个 CPU 准备一个不同的内核栈呢？使用 “即使加上了 big kernel lock，共享内核栈依然会出错” 来描述原因。

Answer

2. 因为在进入内核之前也可能对内核栈进行了操作，例如 trap 之前向内核中压入寄存器信息等。

Answer

4.1.2 Round-Robin Scheduling

这一节的任务是修改内核使之能够进行轮询调度。轮训调度按照如下规则工作：

- kern/shed.c 中的 sched_yield 函数负责选择一个新的环境运行。它按圈顺序搜索 envs[] 数组（如果之前没有运行的环境则选择数组的开头），选择第一个有 ENV_RUNNABLE 状态的环境，并调用 env_run() 跳转到那个环境中。
- sched_yield() 不能同时在两个 CPU 上运行同一个环境。如果环境已经在某一个 CPU 上运行，则其状态会变为 ENV_RUNNING。
- 程序中已经实现了一个新的系统调动 sys_yield()，用户进程可以调用它来放弃 CPU 使用权限。

Exercise 6

在 sched_yield() 中实现一个轮询式调度，并修改 syscall() 来分发 sys_yield()。
确保在 mp_main() 中调用了 sched_yield()。
修改 kern/init.c 来创建三个或更多的环境来运行用户程序 user/yield.c
运行 make qemu，应该能看到环境在结束前来回切换 5 次，输出如下。
使用多个 CPU 进行测试：make qemu CPUS=2

```
...
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

在 yield 程序退出后，将会没有可运行的环境。调度器应该调用 jOS kernel monitor。如果没有发生则表明需要修改代码。

Exercise 6 实验过程

首先在 kern/sched.c 中实现轮询调度，按照题目描述使用按圈轮询的方式，实现的调度代码如下

```
1  void sched_yield(void) {
2      int counter;
```

```
    if (curenv) {
        for (counter = ENVX(curenv->env_id) + 1;
                counter != ENVX(curenv->env_id);
                counter = (counter + 1) % NENV)
            if (envs[counter].env_status == ENV_RUNNABLE)
                env_run(envs + counter);
    } else {
        for (counter = 0; counter < NENV; ++counter)
            if (envs[counter].env_status == ENV_RUNNABLE)
                env_run(envs + counter);
    }

    // sched_halt never returns
    sched_halt();
}
```

然后，在 syscall 中添加一个 case 来使用 sys_yield 系统调用：

```
case SYS_yield:
    sys_yield();
    return 0;
```

最后，修改 kern/init.c 中的用户进程。在 sched_yield(); 前添加：

```
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
```

修改完成后，重新编译，运行 make qemu CPUS=2，结果如图4.5，程序切换 5 次后退出，与预想的一致。

Exercise 6 实验过程

Question

3. 在的 env_run 实现中应该调用了 lcr3()。在调用 lcr3 之前和之后代码进行了对于变量 env_run 的参数 e 的引用。在加载%cr3 寄存器时，MMU 使用的地址环境被改变了。但是虚拟地址的意义与给定的地址上下文有关联：地址上下文指定了虚拟地址映射到的物理地址。那么指针为什么在地址切换前后都能够正常解引用?
4. 任何时候，如果内核从一个环境切换到了另一个，它必须保证老的内核环境寄存器被保存了，以保证她们能够被正确的还原。为什么？这在那里发生的?

Answer

3. 在 lab3 中 env_setup_vm 以内核的页目录作为模板初始化了用户环境。因此两个环境的 e 映射到了同一个物理地址，因此才能够正确的解引用。

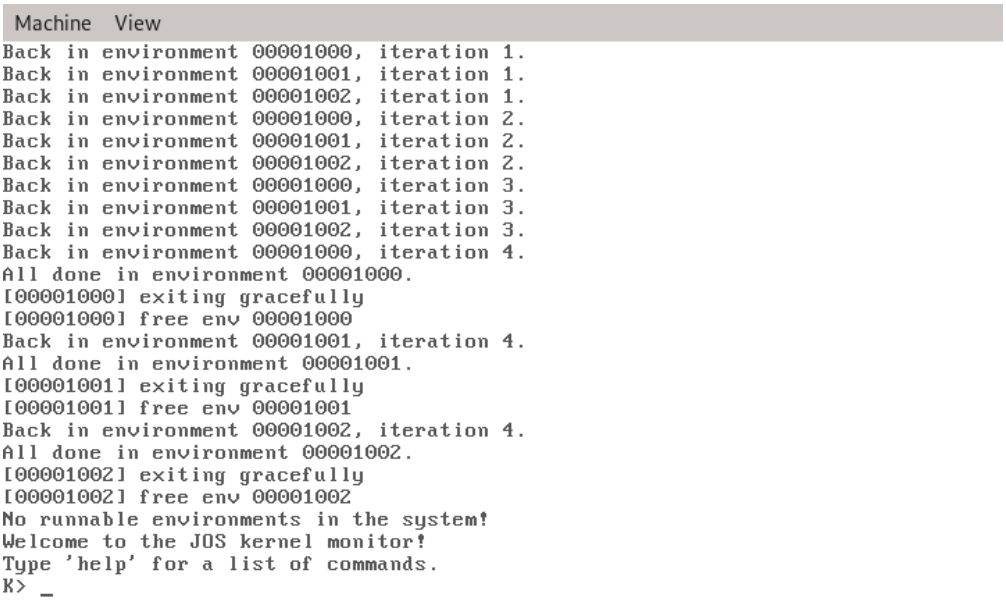

图 4.5: 运行 3 份 user/yield 的输出

4. 保存在 kern/trapentry.S 中的 _alltraps 处，而恢复发生在 kern/env.c 中的 env_pop_tf 处。只有保证内核调用前后栈的内容没有变化才能使用户程序继续正确执行。

Answer

4.1.3 System Calls for Environment Creation

虽然内核可以运行并切换多个用户环境了，但是现在只能运行内核设置好的程序。现在需要实现一个新的系统调用，以允许用户创建并开始新的用户环境。

Unix 提供了 fork 系统调用来创建进程，它会复制父进程的地址空间来创建子进程。唯一的区别是进程 ID。在父进程中，fork 返回子进程的 id，而在子进程中返回 0。父子进程的地址空间是独立的。

现在需要创建一个更原始的 jOS 系统调用来创建一个用户环境。在这个系统调用中需要实现一个类 Unix 的 fork()。加上一些其他的用户环境创建。新的 jOS 系统调用如下：

- sys_exofork:

  这个系统调用创建一个空白进程，在其用户空间中不映射物理内存，并且它是不可运行的。在刚开始它会和父进程有相同的寄存器状态。sys_exofork 还会返回子进程的 envid_t，子进程返回 0。

- sys_env_set_status:

  设置指定进程的状态为 ENV_RUNNABLE 或者 ENV_NOT_RUNNABLE。这个系统调用通常用来标记在地址空间和寄存器状态被初始化之后的新环境就绪。

- sys_page_alloc:

  分配物理地址空间并将其映射到用户环境空间的虚拟地址。

- sys_page_map:

  复制一个用户环境的页映射到另一个用户环境，也就是共享内存。

- sys_page_unmap:

  删除用户环境的虚拟内存映射。

对于所有接收环境 id 的系统调用，jOS 内核支持从值 0 到 “当前用户环境” 的转换。这个转换被 envid2env 实现。

Exercise 7

在 kern/syscall.c 实现上述系统调用。

实现 kern/syscall.c 中的上述系统调用。你需要 kern/pmap.c 以及 kern/env.c 中的不同函数，尤其是 envid2env。现在无论在什么时候调用 envid2env，都需要传递一个 checkperm 参数。使用 user/dumbfork 来测试内核。

Exercise 7 实验过程

首先实现 sys_exofork 函数，该函数分配一个新的进程但是不做内存复制处理。其关键是如何处理父进程返回子进程 id 而子进程返回 0 的问题。于是子进程复制父进程的 TrapFrame 并将 TrapFrame 中 eax 的值设置为 0，而函数本身返回子进程的 id。这样在运行 env_run 时就可以获得不同的返回值了。最后实现的代码如下：

```
1  static envid_t sys_exofork(void) {
2      struct Env *environment;
3      int res;
4      if((res = env_alloc(&environment, curenv->env_id)) < 0)
5          return res;
6      environment->env_status = ENV_NOT_RUNNABLE;
7      environment->env_tf = curenv->env_tf;
8      environment->env_tf.tf_regs.reg_eax = 0;
9      return environment->env_id;
10 }
```

然后是 sys_page_alloc 函数，该函数在环境的目标地址 va 分配一个页面并设置权限为 perm。在实现这个函数时，首先需要检查目标地址以及权限是否合法。根据注释提示，如果 page_insert 失败了还需要释放页面。

```
1  static int sys_page_alloc(envid_t envid, void *va, int perm) {
2      if(!(perm & PTE_U) || !(perm & PTE_U) ||
3              (perm & (~PTE_SYSCALL)) ||
4              va > (void *)UTOP ||
5              va != ROUNDDOWN(va, PGSIZE))
6          return -E_INVAL;
7      struct PageInfo *pginfo = page_alloc(ALLOC_ZERO);
8      if(!pginfo)
9          return -E_NO_MEM;
10     struct Env *environment;
```

```
11      if(envid2env(envid, &environment, 1) < 0)
12          return -E_BAD_ENV;
13      if(page_insert(environment->env_pgdir, pginfo, va, perm) < 0){
14          page_free(pginfo);
15          return -E_NO_MEM;
16      }
17      return 0;
18  }
```

接下来是 sys_page_map 函数。简单的来说，这个函数就是建立跨进程的映射。较为重要的是根据注释对于参数逐个进行检查以及建立正确的映射。

```
1   static int
2   sys_page_map(envid_t srcenvid, void *srcva,
3            envid_t dstenvid, void *dstva, int perm) {
4       if((uint32_t)srcva >= UTOP || PGOFF(srcva) ||
5               (uint32_t)dstva >= UTOP || PGOFF(dstva) ||
6               !(perm & PTE_U) || !(perm & PTE_U) ||
7               (perm & (~PTE_SYSCALL)) )
8           return -E_INVAL;
9       struct Env *src_environemt, *dst_environment;
10      if(envid2env(srcenvid, &src_environemt, 1) < 0 ||
11              envid2env(dstenvid, &dst_environment, 1) < 0)
12          return -E_BAD_ENV;
13      pte_t *pte;
14      struct PageInfo *page = page_lookup(src_environemt->env_pgdir, srcva, &pte)
            ;
15      if(!page || (!(*pte & PTE_W) && (perm & PTE_W)))
16          return -E_INVAL;
17      if(page_insert(dst_environment->env_pgdir, page, dstva, perm) < 0)
18          return -E_NO_MEM;
19      return 0;
20  }
```

然后是 sys_page_unmap 函数，用于取消 sys_page_map 建立的映射。

```
1   sys_page_unmap(envid_t envid, void *va) {
2       if((uint32_t)va >= UTOP || PGOFF(va))
3           return -E_INVAL;
4       struct Env *environment;
5       if(envid2env(envid, &environment, 1) < 0)
6           return -E_BAD_ENV;
7       page_remove(environment->env_pgdir, va);
8       return 0;
9   }
```

然后实现 sys_env_set_status 函数，在子进程内存映射结束后设置状态。与上面几个函数相同，需要注意的是对于参数的检查。

```
1  static int
2  sys_env_set_status(envid_t envid, int status) {
3      if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
4          return -E_INVAL;
5      struct Env *environment;
6      if(envid2env(envid, &environment, 1) < 0)
7          return -E_BAD_ENV;
8      environment->env_status = status;
9      return 0;
10 }
```

最后，需要在 syscall 函数中添加新的系统调用的分发，否则不能正常执行这些系统调用，即在 switch 中添加如下几行：

```
1  case SYS_exofork:
2      return sys_exofork();
3  case SYS_env_set_status:
4      return sys_env_set_status(a1, a2);
5  case SYS_page_alloc:
6      return sys_page_alloc(a1, (void *) a2, a3);
7  case SYS_page_map:
8      return sys_page_map(a1, (void *) a2, a3, (void *) a4, a5);
9  case SYS_page_unmap:
10     return sys_page_unmap(a1, (void *) a2);
```

运行 make run-dumbfork，输出如图4.6所示。父进程在 10 次迭代以后退出，而子进程在 20 次迭代以后退出。

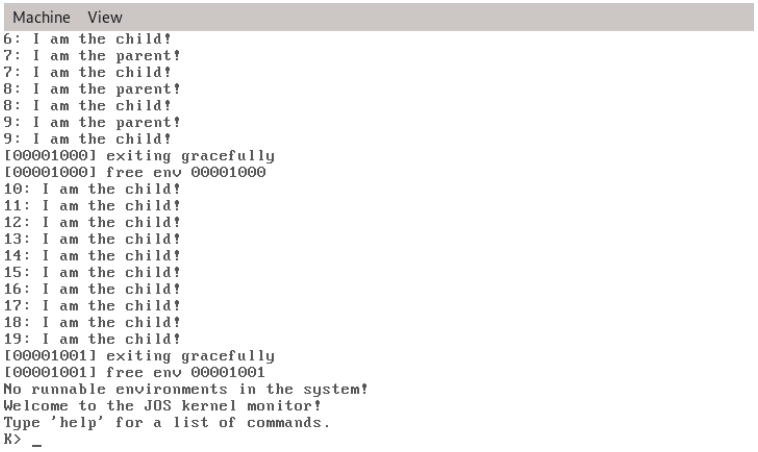

图 4.6: 运行 make run-dumbfork 结果

Exercise 7 实验过程

至此第一部分结束，运行 make grade 以后第一部分能够正常通过，如图4.7所示。

```
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/husixu/Homework/jOS/jos'
dumbfork: OK (2.8s)
Part A score: 5/5

faultread: OK (1.6s)
faultwrite: OK (1.6s)
faultdie: FAIL (1.7s)
    AssertionError: ...
```

图 4.7: 运行 make grade 的部分结果

4.2 Copy-on-Write Fork

Part A 中实现的类 Unix 系统调用较为原始。xv6 将父进程的页复制新的进程页来创建子进程。而这个复制的过程则是整个 fork 最为昂贵的操作。然而调用了 fork 之后子进程通常会调用 exec() 将子进程的内存更换为新的程序，在这种情况下复制父进程的内存这个操作就被浪费了。

因此后来的 Unix 系统让父子进程桐乡同一片物理区域，直到某个进程修改了内存。这个技术叫做 copy-on-write。要实现它就让 fork() 只复制页面的映射关系而补复制内容，同时将页面标记为只读。这样当父进程或子进程写入这个内存值会触发 page fault。这时 Unix 系统才会分配一个新的可写内存给这个进程。这个操作让连续的 fork 和 exec 操作开销大大降低，因为在执行 exec 之前只需要复制一个页面。

在这一部分中将要实现一个 “正确” 的类 Unix 的 fork，实现 copy-on-write。在用户空间实现 copy-on-write 支持使得内核维持简洁和正确。

4.2.1 User-level page fault handling

用户级的 copy-on-write fork 需要知道向写保护的页写入时的 page fault。通常需要设置一个用户空间，这样 page fault 就能指示何时需要进行一些操作。例如，大多数 Unix 内核最初只映射新进程的堆栈区域中的单个页面，随着进程的堆栈消耗增加，并在尚未映射的堆栈地址上发生页面错误，并按需分配额外的页面。例如一个栈的 page fault 会分配并映射一个新的页。一个 BSS 的也错误会分配一个新的页，初始化为 0，再映射。

有很多需要内核跟踪的信息。与传统 Unix 方法不同，在 jOS 中设计者将决定如何处理用户空间中的页面错误。这种设计方式允许程序定义存储区时有较大的灵活性。

Setting the Page Fault Handler

为了处理自己的页面错误用户环境需要在 jOS 注册一个 page fault handler entrypoint。用户环境通过 sys_env_set_pgfault_upcall 来注册自己的 entrypoint，并在 Env 结构中增加 env_pgfault_upcall 来记录这一信息。

Exercise 8

实现 sys_env_set_pgfault_upcall 系统调用。确保在查找环境 id 时使用了权限检查，因为这是一个 “危险” 的系统调用。

Exercise 8 实验过程

按照说明查找用户环境，并实现系统调用即可。实现的代码如下。最后，需要在 syscall 中添加用于分发的 case。

```
1 static int
2 sys_env_set_pgfault_upcall(envid_t envid, void *func) {
3     struct Env *environment;
4     if(envid2env(envid, &environment, 1))
5         return -E_BAD_ENV;
6     environment->env_pgfault_upcall = func;
7     return 0;
8 }
```

在 syscall 中添加的分发代码如下：

```
1 case SYS_env_set_pgfault_upcall:
2     return sys_env_set_pgfault_upcall(a1, (void *) a2);
```

Exercise 8 实验过程

Normal and Exception Stacks in User Environments

在正常运行时，jOS 的进程会运行在正常用户栈上，ESP 在开始指向 USTACKTOP，栈向下增长，数据存放在 USTACKTOP - PGSIZE 到 USTACKTOP - 1 中。当出现页错误时，内核会把进程在一个新的栈上重启，并运行指定的用户级页面错误处理函数，这个栈称为用户异常栈。这个过程完成了进程的栈切换，与从用户态陷入内核态相似。

jOS 的用户异常栈也是一个页这么大，其顶部为虚拟地址 UXSTACKTOP，因此用户异常栈的有效字节在 UXSTACKTOP - PGSIZE 到 UXSTACKTOP - 1 之间。当在这个用户异常栈上运行时，用户页面错误处理程序可以使用 jOS 的普通系统调用来映射页面或调整映射，以解决由页面错误导致的问题。然后用户页面错误处理程序通过汇编语言返回原始栈上的错误代码。

Invoking the User Page Fault Handler

现在需要修改 kern/trap.c 来支持用户级别的页错误处理。如果没有 page fault handler，jOS 会直接销毁用户环境，否则内核会初始化一个 TrapFrame 来记录寄存器的状态，在异常栈上处理页面错误。在 inc/trap.h 中，UTrapframe 如下：

```
1 <-- UXSTACKTOP
2 trap-time esp
3 trap-time eflags
4 trap-time eip
5 trap-time eax       start of struct PushRegs
6 trap-time ecx
7 trap-time edx
8 trap-time ebx
9 trap-time esp
```

```
10  trap-time ebp
11  trap-time esi
12  trap-time edi       end of struct PushRegs
13  tf_err (error code)
14  fault_va            <-- %esp when handler is run
```

内核会重新安排用户环境并开始执行。fault_va 是导致页面错误的虚拟地址。如果用户环境在发生异常时已经在用户异常栈上运行，则页面错误处理程序应该出错，在这种情况下应该在 tf->tf_esp 而不是 UXSTACKTOP 上构造新管道栈。应该先压入一个 32 位字，然后才是 UTrapframe。

要测试 tf->tf_esp 是否在用户异常栈中，需要检查它是否在 UXSTACKTOP - PGSIZE 到 UXSTACKTOP - 1 的范围。

Exercise 9

实现 kern/trap.c 中的 page_fault_handler 来分发页面错误到用户模式的处理函数。确保在写入异常栈时使用了正确的措施。

Exercise 9 实验过程

在实现用户异常栈调用时，需要考虑这个 page fault 是否由另一个 page fault 引起，如果是的话需要递归式上行调用。也就是说，需要将 page fault 分为两种情况：

- 用户进程发生 page fault，此时栈的切换顺序为用户栈 → 内核栈 → 用户异常栈。
- 在 page fault 时又发生了 page fault。此时已经在用户异常栈了，但是还是会重复一遍上面的步骤，此处压入 UTrapframe 需要留 4 个字节的空位，而栈的切换顺序为用户异常栈 → 内核栈 → 用户异常栈。

此外还需要考虑权限以及地址是否用完等问题，因此在 page_fault_handler 中添加的代码如下：

```
1  if(curenv->env_pgfault_upcall){
2      struct UTrapframe *utf;
3      uintptr_t addr;         // addr of utf
4      if(USTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < USTACKTOP)
5          addr = tf->tf_esp - sizeof(struct UTrapframe) - 4;
6      else
7          addr = UXSTACKTOP - sizeof(struct UTrapframe);
8      user_mem_assert(curenv, (void *)addr, sizeof(struct UTrapframe), PTE_W);
9      utf = (struct UTrapframe *)addr;
10     utf->utf_fault_va = fault_va;
11     utf->utf_err = tf->tf_err;
12     utf->utf_regs = tf->tf_regs;
13     utf->utf_eip = tf->tf_eip;
14     utf->utf_eflags = tf->tf_eflags;
15     utf->utf_esp = tf->tf_esp;
16
17     tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
18     tf->tf_esp = addr;
```

```
19      env_run(curenv);
20  }
```

Exercise 9 实验过程

User-mode Page Fault Entrypoint

接下来需要实现汇编程序来调用 C 的页面错误处理程序，并在发生错误的地方恢复执行。这个汇编程序就是用 sys_env_set_pgfault_upcall() 向内核注册的处理程序。

Exercise 10

实现 lib/pfentry.S 中的 _pgfault_upcall。有趣的是讷河将会直接返回到导致页面错误的用户代码中，而不需要通过内核。同时切换堆栈并重加载 EIP 是较为困难的。

Exercise 10 实验过程

将返回的错误地址填到之前预先保留的 4 个字节处，然后在恢复寄存器，从而达到同时切换 esp 和 eip 的效果。实现的代码如下：

```
1  .text
2  .globl _pgfault_upcall
3  _pgfault_upcall:
4      // Call the C page fault handler.
5      pushl %esp          // function argument: pointer to UTF
6      movl _pgfault_handler, %eax
7      call *%eax
8      addl $4, %esp           // pop function argument
9
10     movl 0x28(%esp), %eax
11     subl $0x4, 0x30(%esp)
12     movl 0x30(%esp), %edx
13     movl %eax, (%edx)
14     addl $0x8, %esp
15
16     popal
17
18     addl $0x4, %esp
19     popfl
20
21     popl %esp
22
23     ret
```

Exercise 10 实验过程

Exercise 11

完成 lib/pgfault.c 中的 set_pgfault_handler() 代码。

Exercise 11 实验过程

这一函数是用户用于指定缺页异常处理方式的函数，需要补充的代码用于初始化 _pgfault_upcall 和 _pgfault_handler，其中 _pgfault_upcall 就是 lib/pgentry.S 中的上行调用入口，_pgfault_handler 在 _pgfault_upcall 中被调用。补充完整后的代码如下：

```
1  void set_pgfault_handler(void (*handler)(struct UTrapframe *utf)) {
2      int r;
3      if (_pgfault_handler == 0) {
4          envid_t envid = sys_getenvid();
5          if(sys_page_alloc(envid, (void *)(UXSTACKTOP-PGSIZE), PTE_U | PTE_W |
               PTE_P) < 0)
6              panic("set_pgfault_handler: sys_page_alloc failed. ");
7          if(sys_env_set_pgfault_upcall(thisenv->env_id, _pgfault_upcall) < 0)
8              panic("set_pgfault_handler: sys_env_set_pgfault_upcall failed.");
9      }
10     _pgfault_handler = handler;
11 }
```

Exercise 11 实验过程

4.2.2 Implementing Copy-on-Write Fork

现在我们已经有了在用户控件实现 copy-on-write fork 的条件。在 lib/fork.c 中提供了 fork 的框架。像 dumbfork 一样，fork 应该能够创建一个新的环境，然后扫描父进程的整个地址空间并创建对应的子进程空间的调用。不同的是，dumbfork 复制了整个页而 fork 只会创建映射。fork 的基本流程如下：

1. 父进程将 pgfault 设置为 page fault 的处理函数。
2. 父进程使用 sys_exofork 创建一个子进程。
3. 对于每一个 UTOP 之下可写的页面以及标记了 copy-on-write 的页面，父进程使用 suppage 将其映射到子进程，同时将权限修改为只读。

   异常栈有不同的分配方式。需要在子进程中分配一个新的页面，因为 page fault handlerhi 向异常栈中写入内容并在异常栈上运行。如果异常栈页面用 copy-on-write，就不能执行复制过程了。
4. 父进程会为子进程设置 user page fault entrypoint。
5. 子进程现在已经就绪，父进程将其标记为 runnable。

Exercise 12

实现 lib/fork.c 中的 fork, duppage 以及 pgfault。

使用 forktree 测试代码，这个程序除了 new env, free env 和 exiting gracefully 以外应该产生如下信息（有可能不是以这个顺序产生）：

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

Exercise 12 实验过程

首先实现 fork。参考 user/dumbfork.c 中的 dumbfork，但是需要修改的是设置 page fault handler，且 fork 不需要复制内核映射，此外还需要为子进程设置 user page fault entrypoint。实现后的 fork 如下：

```
1  envid_t fork(void) {
2      // LAB 4: Your code here.
3      set_pgfault_handler(pgfault);
4      envid_t envid = sys_exofork();
5      if (envid < 0)
6          panic("fork: sys_exofork failed.");
7      if (envid == 0) {
8          thisenv = envs + ENVX(sys_getenvid());
9          return 0;
10     }
11
12     uint32_t addr;
13     for (addr = 0; addr < USTACKTOP; addr += PGSIZE)
14         if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P) )
15             duppage(envid, PGNUM(addr));
16
17     if (sys_page_alloc(envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W |
           PTE_P) < 0)
18         panic("fork: sys_page_alloc failed.");
19
20     sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
```

```
21
22      if (sys_env_set_status(envid, ENV_RUNNABLE) < 0)
23          panic("fork: sys_env_set_status failed.");
24
25      return envid;
26  }
```

然后实现 duppage 函数，用于复制父子用户环境之间的页面映射。

```
1   static int duppage(envid_t envid, unsigned pn) {
2       int r;
3       void *addr = (void *)(pn * PGSIZE);
4       if((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)){
5           if(sys_page_map(0, addr, envid, addr, PTE_U | PTE_COW | PTE_P) < 0)
6               panic("duppage: parent->child sys_page_map failed.");
7           if(sys_page_map(0, addr, 0, addr, PTE_U | PTE_COW | PTE_P) < 0)
8               panic("duppage: self sys_page_map failed.");
9       } else {
10          if(sys_page_map(0, addr, envid, addr, PTE_P | PTE_U) < 0)
11              panic("duppage: single sys_page_map failed.");
12      }
13      return 0;
14  }
```

最后实现 _pgfault_upcall 中调用的页面错误处理函数。在 diayon 在调用前父子进程的错误地址都在同一个物理内存，而这一函数的作用是分配一个新的物理页面使得两者独立。最终实现的函数如下：

```
1   static void pgfault(struct UTrapframe *utf) {
2       void *addr = (void *) utf->utf_fault_va;
3       uint32_t err = utf->utf_err;
4       int r;
5
6       if(!(err & FEC_WR) || !(uvpt[PGNUM(addr)] & PTE_COW))
7           panic("pgfault: invalid UTrapFrame");
8
9       envid_t envid = sys_getenvid();
10      addr = ROUNDDOWN(addr, PGSIZE);
11      if(sys_page_alloc(envid, PFTEMP, PTE_U | PTE_W | PTE_P) < 0)
12          panic("pgfault: sys_page_alloc failed.");
13      memcpy(PFTEMP, addr, PGSIZE);
14      if(sys_page_map(envid, PFTEMP, envid, addr, PTE_U | PTE_W | PTE_P) < 0)
15          panic("pgfault: sys_page_map failed.");
16      if(sys_page_unmap(envid, PFTEMP) < 0)
17          panic("pgfault: sys_page_unmap failed.");
18  }
```

Exercise 12 实验过程 。

4.3 Preemptive Multitasking and Inter-Process communication (IPC)

4.3.1 Clock Interrupts and Preemption

运行 user/spin，这个测试程序运行一个子环境，获得 CPU 控制权后它会死循环，父环境和内核都不会获得 CPU。这显然不利于保护内核免受用户环境的影响。因此需要允许内核抢占正在运行的环境并强行重新获得 CPU 控制权。此时需要扩展 jOS 以支持来自时钟的硬件中断。

Interrupt discipline

外部中断 (IRQ) 一共由 16 种，在 picirg.c 中将 IRQ_OFFSET 映射到了 IDT。在 inc/trap.h 中 IRQ_OFFSET 被定义为 32，因此 IDT[32] 包含了时钟中断的入口地址。

在 jOS 中，相对于 xv6 做了一个关键的简化：在内核态禁用外部中断。外部中断使用%eflag 寄存器的 FL_IF 位控制。当该位为 1 时开启中断。由于 jOS 的这一简化，只需要在进入以及离开内核时需要对这个位进行修改。

内核需要确保在用户态时 FL_IF 为 1，从而使得有中断发生时内核能够处理。bootloader 的第一条指令 cli 关闭了中断，从那之后就再未开启过。

Exercise 13

修改 kern/trapentry.S 以及 kern/trap.c 来初始化 IDT 表项并为 IRQ0~15 提供处理函数。然后修改 kern/env.c 中的 env_alloc() 来保证用户环境总是中断使能的。同样，上述 sched_halt() 中的 sti 指令的注释，使得空闲 CPU 不屏蔽中断。

Exercise 13 实验过程

参考 lab3 中设置 trap 的方法，在 kern/trapentry.S 中加入：

```
1 TRAPHANDLER_NOEC(handler_timer,    IRQ_OFFSET + IRQ_TIMER)
2 TRAPHANDLER_NOEC(handler_kbd,      IRQ_OFFSET + IRQ_KBD)
3 TRAPHANDLER_NOEC(handler_serial,   IRQ_OFFSET + IRQ_SERIAL)
4 TRAPHANDLER_NOEC(handler_spurious, IRQ_OFFSET + IRQ_SPURIOUS)
5 TRAPHANDLER_NOEC(handler_ide,      IRQ_OFFSET + IRQ_IDE)
6 TRAPHANDLER_NOEC(handler_error,    IRQ_OFFSET + IRQ_ERROR)
```

并在 kern/trap.c 中加入 6 个函数的声明，然后在 trap_init() 中加入如下代码：

```
1 SETGATE(idt[IRQ_OFFSET + IRQ_TIMER],    0, GD_KT, handler_timer,    0);
2 SETGATE(idt[IRQ_OFFSET + IRQ_KBD],      0, GD_KT, handler_kbd,      0);
3 SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL],   0, GD_KT, handler_serial,   0);
4 SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, handler_spurious, 0);
5 SETGATE(idt[IRQ_OFFSET + IRQ_IDE],      0, GD_KT, handler_ide,      0);
6 SETGATE(idt[IRQ_OFFSET + IRQ_ERROR],    0, GD_KT, handler_error,    0);
```

最后，在 kern/env.c 的 env_alloc() 中加入 e->env_tf.tf_eflags |= FL_IF; 使能用户中断，以使得中断发生时内核能够正确处理，并取消 kern/sched.c 中 sti 的注释。

Exercise 13 实验过程

Handling Clock Interrupts

在 user/spin 程序中，子进程开启以后就陷入死循环，此后 kernel 无法再次获得控制权。我们需要让硬件周期性地产生时钟中断，将控制权交给 kernel，使得我们能够切换到其他的进程。

> **Exercise 14**
>
> 修改内核的 trap_dispatch 函数，让它调用 sched_yield() 来在时钟中断发生时查找并运行一个不同的环境。

Exercise 14 实验过程

要处理时钟中断，只需要在 trap_dispatch 中添加如下代码即可。注意需要使用 lapic_eoi() 来接受中断。

```
1 if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
2     lapic_eoi();
3     sched_yield();
4 }
```

Exercise 14 实验过程

4.3.2 Inter-Process communication (IPC)

前几部分一致在关注操作系统的隔离问题，这造成了每个程序都拥有整个 CPU 的错觉。操作系统的一个重要的功能是允许服务程序在彼此需要是进行通讯。进程间通讯有很多模型，但在 jOS 中只会实现一个简单的 IPC 机制。

IPC in JOS

这一部分将实现两个系统调用 sys_ipc_recv 以及 sys_ipc_try_send，然后将他们封装为 2 个函数库，并通过 ipc_recv 以及 ipc_send 进行通信。IPC 的消息由两部分组成：一个 32 位值以及一个映射。

Sending and Receiving Messages

要接收消息，调用函数 sys_ipc_recv。这一系统调用取消环境的调度，并在接收到消息之前一直阻塞，此时任何其他环境都可以向其发送消息。也就是说 Part A 中的权限检查不适用于 IPC，因为 IPC 的系统调用是安全的。

要发送消息，需要调用函数 sys_ipc_try_send。如果指定的环境正在接收消息，则将发送消息并返回 0，否则返回-E_IPC_NOT_RECV 来标识目标环境不在接收消息。库函数 ipc_send 将重复调用 sys_ipc_trap_send 直到发送成功。

Transferring Pages

当进程调用 sys_ipc_recv 并踢动 dstva 虚拟地址时，用户环境表明它愿意接收页面映射。如果发送者发送了一个页面，那么页面应该映射到 dstva 中。如果接受者已经有了一个映射到 dstva 的页面，那么这一页面就不会映射。

当环境调用 sys_ipc_try_send 并使用 srcva 作为参数时，表明发送者想要将映射到 srcva 的页面以 perm 的权限发送给接收者。在一个成功的 IPC 之后，发送方保留原页面的映射，接收方在指定的地址空间获得相同的页面。

如果发送方或接收方指示应该传递页面，则不传递页面。IPC 之后内核将接收方的 Env 中的 env_ipc_perm 设置为接收到的页面的权限，如果没有收到则为 0。

Implementing IPC

Exercise 15

实现 kern/syscall.c 中的 sys_ipc_recv 以及 sys_ipc_try_send。实现之前先阅读注释，因为它们需要共同工作。当调用 envid2 时，应该设置 checkperm 为 0，这意味着任何环境都被允许发送 IPC 消息到任何环境，并且内核除了检查 envid 是否有效外不做特殊的权限检查。然后实现 lib/ipc.c 中的 ipc_recv 以及 ipc_send。

Exercise 15 实验过程

首先实现 sys_ipc_recv。需要注意的是，如果参数指示的虚拟地址大于 UTOP 时不能报错退出，而需要忽略，因为这只说明接受者只需要接收值而不需要共享页面。实现的代码如下：

```
1  static int sys_ipc_recv(void *dstva) {
2      if((uint32_t)dstva < UTOP && PGOFF(dstva))
3          return -E_INVAL;
4      curenv->env_ipc_recving = true;
5      curenv->env_ipc_dstva = dstva;
6      curenv->env_status = ENV_NOT_RUNNABLE;
7      sys_yield();
8      return 0;
9  }
```

接下来实现 sys_ipc_try_send，需求与 sys_page_map 类似，但是不能直接调用 sys_page_map，因为 sys_page_map 会检查 env 的权限。实现的 sys_ipc_try_send 如下：

```
1  static int sys_ipc_try_send(envid_t envid,
2          uint32_t value, void *srcva, unsigned perm) {
3      struct Env *env;
4      if(envid2env(envid, &env, 0) < 0)
5          return -E_BAD_ENV;
6      if(!env->env_ipc_recving)
7          return -E_IPC_NOT_RECV;
8
9      pte_t *pte;
```

```
10      int res;
11      struct PageInfo *page = page_lookup(curenv->env_pgdir, srcva, &pte);
12      if((uint32_t)srcva < UTOP && (
13                  PGOFF(srcva) || !page ||
14                  (perm & PTE_W) != (*pte & PTE_W) ||
15                  !(perm & PTE_U) || !(perm & PTE_P) ||
16                  (perm & (~PTE_SYSCALL))) )
17              return -E_INVAL;
18      if((uint32_t)srcva < UTOP && (res = page_insert(env->env_pgdir, page, env->
            env_ipc_dstva, perm)) < 0)
19          return res;
20
21      env->env_ipc_recving = 0;
22      env->env_ipc_from = curenv->env_id;
23      env->env_ipc_perm = perm;
24      env->env_ipc_value = value;
25      env->env_status = ENV_RUNNABLE;
26      return 0;
27  }
```

接下来实现对于上面两个系统调用的封装。首先是 ipc_recv，值得注意的是当不需要共享页面时将虚拟地址设置为大于或等于 UTOP 的值。ipc_recv 实现的代码如下：

```
1   int32_t ipc_recv(envid_t *from_env_store, void *pg, int *perm_store) {
2       if(!pg)
3           pg = (void *)UTOP;
4       int res;
5       if((res = sys_ipc_recv(pg)) < 0){
6           if(from_env_store)
7               *from_env_store = 0;
8           if(perm_store)
9               *perm_store = 0;
10          return res;
11      }
12      if(from_env_store)
13          *from_env_store = thisenv->env_ipc_from;
14      if(perm_store)
15          *perm_store = thisenv->env_ipc_perm;
16      return thisenv->env_ipc_value;
17      return 0;
18  }
```

然后是 ipc_send，同样，当 pg 为 NULL 时设置虚拟地址为大于或等于 UTOP 的值。ipc_send 实现如下：

```
1   void ipc_send(envid_t to_env, uint32_t val, void *pg, int perm) {
2       if(!pg)
```

```
3           pg = (void *)UTOP;
4       int res;
5       while(1){
6           res = sys_ipc_try_send(to_env, val, pg, perm);
7           if(!res)
8               return;
9           if(res != -E_IPC_NOT_RECV)
10              panic("ipc_send: not receiving");
11          sys_yield();
12      }
13  }
```

最后，将这两个 syscall 加入 syscall 函数分支中：

```
1  case SYS_ipc_try_send:
2      return sys_ipc_try_send(a1, a2, (void *)a3, a4);
3  case SYS_ipc_recv:
4      return sys_ipc_recv((void *)a1);
```

Exercise 15 实验过程

至此，Lab4 全部完成，运行 make grade，能够通过全部测试，如图 4.8所示。

```
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/husixu/Homework/jOS/jos'
dumbfork: OK (1.4s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (1.1s)
faultdie: OK (1.1s)
faultregs: OK (1.0s)
faultalloc: OK (1.1s)
faultallocbad: OK (1.6s)
faultnostack: OK (1.5s)
faultbadhandler: OK (1.5s)
faultevilhandler: OK (1.0s)
forktree: OK (1.3s)
Part B score: 50/50

spin: OK (1.1s)
stresssched: OK (2.7s)
sendpage: OK (1.1s)
pingpong: OK (1.1s)
primes: OK (10.0s)
Part C score: 25/25

Score: 80/80
 ~/H/j/jos  lab4                33s  Sat Mar  3 19:30:18 2018
```

图 4.8: 运行 make grade 输出结果