[TOC]

OS_lab1学习笔记

练习2：

首先声明练习2的文档是在参考了https://www.cnblogs.com/fatsheep9146/p/5078179.html 的情况下写的，但是对于原博客有些地方某些没解释的地方加了查阅的资料和自己的理解，对原文有误以及不认同的地方进行了修改删除。

运行gdb后出现的第一条指令是

```
1. 0xffff0:     ljmp $0xf000,$0xe05b
```

执行一个长跳转，通过段地址左移4位加上便宜地址计算出应该跳转到`0xfe05b`地址处。

```
2. 0xfe05b:     cmpl $0x0,%cs:0x65b4
```

是把0和`$cs`所代表的内存地址处的值相比较，具体为了实现什么并不清楚。

```
3. 0xfe062:     jne 0xfd3aa
```

即上一条指令结果不为0，也就是`$cs:0x65b4`处的值不为0的时候跳转到`0xfd3aa`处，同样也不知道具体是为什么。

```
4. 0xfe066:     xor    %ax,%ax
```

指令的地址增加4，说明`$cs:0x65b4`处的值为0没执行跳转，该条指令将ax寄存器置为0。

```
5. 0xfe068:     mov    %ax,%ss
6. 0xfe06a:     mov    $0x7000,%esp
7. 0xfe070:     mov    $0xf431f,%edx
```

将ss(堆栈段寄存器)清零,给esp(栈指针寄存器)，edx赋值。

```
8. 0xfe076:     jmp    0xfd233
9. 0xfd233:     mov    %eax,%ecx
```

强制跳转到0xfd233处，使ecx和eax都为0。

```
10.  0xfd236:   cli
```

cli是汇编禁止中断指令，只能在内核态下使用，保证了在启动的时候不会被其他硬件所干扰。

```
11.  0xfd237:   cld
```

清除方向标志，在字符串的比较，赋值，读取等一系列操作中，di或si是可以自动增减的而不需要人来加减它的值，cld即告诉程序si，di向前移动，与之对应的std指令为设置方向，告诉程序si，di向后移动。

```
12.  0xfd238:   mov    $0x8f,%eax
13.  0xfd23e:   out    %al,$0x70
14.  0xfd240:   in     $0x71,%al
```

out和in是对I/O端口的读写操作的汇编指令，因为x86CPU没有对I/O端口和内存进行统一编址，所以对I/O端口的读写就没办法像对内存读写一样使用mov等操作指令，否者出现相同的地址时CPU没办法单纯通过指令来区分出到底应该对内存还是外设进行操作。而且似乎只能通过al,ax,eax来传递和I/O端口有关的数据，至于为什么查了一圈没有找到，可能是规定好了吧。

通过这个链接http://bochs.sourceforge.net/techspec/PORTS.LST 能看到具体某个端口的功能是什么。（下文加粗的内容直接摘自文档）例如**0070-007F ---- CMOS RAM/RTC (Real Time Clock MC146818)**,说明0x70-7f端口是控制CMOS RAM/RTC的。

CMOS内存中保存着系统断电时的数据，用来记录软盘的类型和数目、硬盘大小信息、内存大小以及其他重要信息。而在CMOS芯片中还含有一个实时时钟(RTC),用来保存当前时间。关掉电源时，RTC(Real Time Clock)由计算机内部的电池供电，保持时钟处于活跃状态，以及保留CMOS内存中的内容。在BIOS系统上电自检的时候会读取RTC的时间作为系统时间。

查看文档可知0x70端口的最高位是NMI的使能端**(bit 7 = 1 NMI disabled = 0 NMI enabled)，剩下的**6**位则是用来访问**CMOS**内存的(bit 6-0 CMOS RAM index)。个人理解**0x71**端口应该是**0x70**端口后**7**位的地址所在的内存中存的数据，文档中关于**0x71**端口内容没太看懂。文档中还提到了在向**0x70**端口写入值后，必须对**0x71**端口有操作**(**这里正好选用了**in**操作**)**，否则**RTC**就会处于不确定的状态。（any write to 0070 should be followed by an action to 0071 or the RTC wil be left in an unknown state.）**

至于NMI(Non-Maskable Interrupt)，即不会被CPU屏蔽的中断，用来通知CPU发生了灾难性事件，如电源掉电、存储器读写出错、总线奇偶位出错等。

该段指令把eax赋值为0x8f，al也就是0b0111 1111，接着下一条指令将al的值写入了0x70端口，给0x70最高位传送了0，保证了在启动时不会被NMI打断，避免出错。接下来再从0x71端口读出数据给al。

```
15.  0xfd242:   in     $0x92,%al
16.  0xfd244:   or     $0x2,%al
17.  0xfd246:   out    %al,$0x92
```

0x92端口是PS/2系统的控制端口**(PS/2 system control port A (port B is at 0061)),后两条指令其实就是把该端口的倒数第二位置为**1**，而bit 1 = 1 indicates A20 active**，该步是系统为了由实系统切换到保护系统做的准备。

```
18.  0xfd248:   lidtw  %cs:0x68f8
19.  0xfd24e:   lgdtw  %cs:0x68b4
```

lidt/lgdt指令，将源操作数中的值加载到中断向量表寄存器 (IDTR)或全局描述符表寄存器 (GDTR)。 该指令把从地址0x68f8起始的后面数据读入到中断向量表寄存器(IDTR)中，把从0x68b4为起始地址处的6个字节的值加载到全局描述符表格寄存器中GDTR中。中断是操作系统中非常重要的一部分，有了中断操作系统才能真正实现进程。每一种中断都有自己对应的中断处理程序，那么这个中断的处理程序的首地址就叫做这个中断的中断向量。（这段指令不太理解）

```
20.  0xfd254:   mov    %cr0,%eax
21.  0xfd257:   or     $0x1,%eax
22.  0xfd25b:   mov    %eax,%cr0
```

计算机中包含CR0~CR3四个控制寄存器，用来控制和确定处理器的操作模式。CR0中含有控制处理器操作模式和状态的系统控制标志，CR1保留不用，CR2含有导致页错误的线性地址，CR3中含有页目录表物理内存基地址，因此该寄存器也被称为页目录基地址寄存器。该段指令将CR0的最低位置为1，CR0寄存器的最低位是启动保护位，当该位被置1，代表进入了保护模式。

```
23.  0xfd25e:   ljmpl  $0x8,$0xfd266
24.  0xfd266:   mov    $0x10,%ax
25.  0xfd269:   add    %al,(%bx,%si)
26.  0xfd26b:   mov    %ax,%ds
27.  0xfd26d:   mov    %ax,%es
28.  0xfd26f:   mov    %ax,%ss
29.  0xfd271:   mov    %ax,%fs
30.  0xfd273:   mov    %ax,%gs
31.  0xfd275:   mov    %cx,%ax
```

经过一个长跳转后就是对各种寄存器进行赋值，仍然是不太懂这么做的含义。

```
23.  0xfd25e:   ljmpl  $0x8,$0xfd266
```

si后显示如下

> The target architecture is assumed to be i386

应该表明切换到了i386，进入了保护模式。

练习3：

代码解析

> boot.s

```
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64
```

通过链接http://bochs.sourceforge.net/techspec/PORTS.LST 可以看到端口0x64的作用

```
0064    r       KB controller read status (ISA, EISA)
                 bit 7 = 1 parity error on transmission from keyboard
                 bit 6 = 1 receive timeout
                 bit 5 = 1 transmit timeout
                 bit 4 = 0 keyboard inhibit
                 bit 3 = 1 data in input register is command 0 data in
input register is data
                 bit 2   system flag status: 0=power up or reset
1=selftest OK
                 bit 1 = 1 input buffer full (input 60/64 has data for
8042)
                 bit 0 = 1 output buffer full (output 60 has data for
system)
```

该段代码在不断检测0x64的bit1位是否为1，不为1就一直循环检测。当bit1位为1时，说明输入缓冲区满了，也就是控制器并没有取走输入缓冲区的数据，所以需要一直等待直到控制器取走相应的数据才会继续执行后面的命令。取走后给0x64端口写入数据。

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

lgdt gdtdesc，是把gdtdesc这个标识符的值送入全局映射描述符表寄存器GDTR中。boot.s最后有对gdtdesc的定义

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL                                # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)       # code seg
  SEG(STA_W, 0x0, 0xffffffff)             # data seg

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

可知这条指令的功能就是把关于GDT表的一些重要信息存放到GDTR寄存器中，其中包括GDT表的内存起始地址，以及GDT表的长度。当加载完GDT表的信息到GDTR寄存器之后紧接着修改CR0寄存器的内容。CR0寄存器的bit0是保护模式启动位，把这一位值1代表保护模式启动。

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

跳转指令，这条指令的目的在于由当前的运行的实模式切换成32位地址的保护模式。

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

经过一系列对寄存器的赋值操作，然后对esp进行赋值，然后跳转到main.c文件中的bootmain函数处（在main.c文件中）。

> main.c

```
        struct Proghdr *ph, *eph;

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

main.c函数开头先调用了readseg函数读取了磁盘第一页

```
void readseg(uint32_t pa, uint32_t count, uint32_t offset)
```

readseg函数定义如下，通过注释来推测函数功能，应该是把距离内核起始地址offset个偏移量存储单元作为起始，将它以及之后count个字节的数据读出送入以pa为起始地址的内存物理地址处。 因此这条指令把内核的第

一页(SECTSIZE*8* = *512*8= 4096 = 4MB)的内容读取的内存地址ELFHDR(0x10000)处。 文章开头对SECTSIZE以及ELFHDR进行了宏定义如下：

```
#define SECTSIZE        512
#define ELFHDR          ((struct Elf *) 0x10000) // scratch space
```

随后判断是否是个空的elf文件

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

于是便去查找elf文件是什么

> 在计算机科学中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。 ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且他们的位置也未必如同所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。 在x86架构下，Linux使用的是ELF(Executable and Linkable Format)目标文件格式。

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

经过一大段看不懂的代码后，通过这段的注释可以看出是通过ELF文件的头部找到某个入口点，并且没有返回，故可以推测是由此开始运行内核文件。

练习

> 比较反汇编的指令和boot.s,boot.asm

在gdb窗口中输入 b *0x7c00，然后再输入c，表示继续运行到断点处，然后输入`x/30i 0x7c00`指令，把存放在0x7c00以及之后的30条指令反汇编出来，直接和boot.s以及在obj/boot/boot.asm进行比较。 这三者在指令上没有区别，只不过在源代码boot.s中，我们定义了很多标识符比如set20.1，.start，这些标识符在被汇编成机器代码后都会被转换成真实物理地址，如set20.1就被转换为0x7c0a。在obj/boot/boot.asm中还列出了对应关系，在真实执行时，就只能看到真实物理地址。

问题解答

> 处理器什么时候开始执行32位代码？如何完成的从16位到32位模式的切换？

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

在运行完该条指令后CPU进入保护系统，也就是32位地址模式。CR0寄存器中含有控制处理器操作模式和状态的系统控制标志，CR0寄存器的最低位是启动保护位，当该位被置1，代表进入了保护模式。

> 引导加载程序bootloader执行的最后一个指令是什么，加载的内核的第一个指令是什么？

boot loader执行的最后一条语句是main.c中bootmain中的最后一条语句

```
(void (*)(void)) (ELFHDR->e_entry))();
```

转到操作系统内核程序处。通过查看/kern/entry.s文件,可以得知加载内核的第一个指令是

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

> 内核的第一个指令在哪里?

内核的第一条指令位于/kern/entry.s文件中（可能是该这样回答吧）

> 引导加载程序如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？

通过 ELF 文件头获取所有 program header table，每个 program header table 记录了三个重要信息用以描述段(segment)：p_pa (物理内存地址)，p_memsz (所占内存大小)，p_offset (相对文件的偏移地址)。根据这三个信息，对每个段，从 p_offset 开始，读取 p_memsz 个 byte 的内容（需要根据扇区(sector)大小对齐），放入 p_pa 开始的内存中。（这个答案是从网上找的答案，并不算是很懂。大概知道是main.c里面的函数实现的这部分功能，具体怎么通过哪些代码来实现不是很懂）

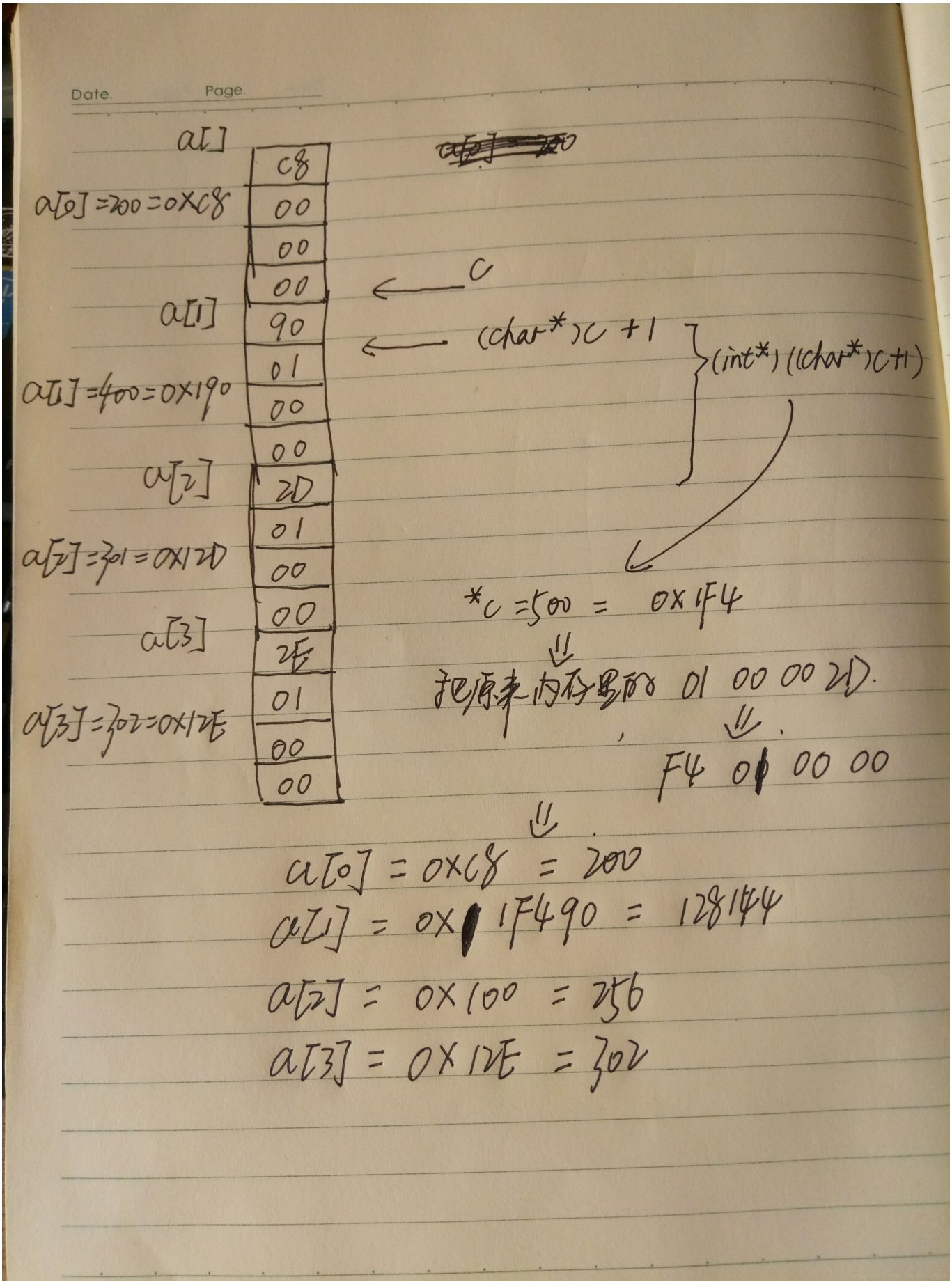
练习4：

```
gcc -o pointers pointers.c
./pointers
```

在终端中进行pointers.c所在的目录并输入这两行命令即可编译并运行，得到输出如下：

```
1: a = 0x7fffbf52e990, b = 0x1f6d010, c = 0x7465675f6f736476
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 = 0x7fffbf52e990, b = 0x7fffbf52e994, c = 0x7fffbf52e991
```

一开始看下来对于`3[c] = 302,`这句以及第五行的输出结果没太懂，后面尝试了3[c]和[c]3，发现结果是一样的，因为觉得这二者可能是等价的，但是具体语法是为什么还不太懂。



第6行的地址是因为int类型是4字节，而char类型是1字节所以不同。

练习**5**：

> 将boot/Makefrag中的链接地址更改为一个错误的地址，运行make clean，用make命令重新编译实验，然后再次跟踪到引导加载程序，看看会发生什么

打开boot/Makefrag文件，看到链接地址为0x7c00

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
        @echo + ld boot/boot
        $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
        $(V)$(OBJDUMP) -S $@.out >$@.asm
        $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
        $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

修改为0x7E00 然后make qemu结果如下：

```
EAX=00000011 EBX=00000000 ECX=00000000 EDX=00000080
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00006f20
EIP=00007c2d EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
CS =0000 00000000 0000ffff 00009b00 DPL=0 CS16 [-RA]
SS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
DS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
FS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
GS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00000000 00000000
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

链接地址是通过编译器链接器处理后形成的可执行文件理论上应该执行的地址，即逻辑地址。加载地址则是可执行文件真正被装入内存后运行的地址，即物理地址。在运行boot loader时，boot loader中的链接地址（逻辑地址）和加载地址（物理地址）是一样的，但是当进入到内核程序后，这两种地址就不再相同了。

练习**6**：

在进入boot loader之前，从内存地址0x00100000处开始之后8个字的内容为：

```
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

在进入boot loader后 在bootmain最后一句call *0x10018的地址0x7d6b处加断点

```
 b *0x7d6b
Breakpoint 11 at 0x7d6b
(gdb) c
Continuing.
=> 0x7d6b:      call   *0x10018

Breakpoint 11, 0x00007d6b in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
```

bootmain函数在最后会把内核的各个程序段送入到内存地址0x00100000处，所以这里现在存放的就是内核的某一个段的内容。

练习7：

> 使用QEMU和GDB跟踪到JOS内核并停止在movl%eax，%cr0。 查看内存中在地址0x00100000和0xf0100000处的内容

在`mov %eax, %cr0`所在的地址0x100025处加断点，运行到此处，使用x/16xw查看两个地址往后16字的内容

```
The target architecture is assumed to be i386
=> 0x100025:    mov    %eax,%cr0

Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/16xw 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
0x100020:       0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
0x100030:       0x00000000      0x110000bc      0x0056e8f0      0xfeeb0000
(gdb) x/16xw 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000
0x00000000
0xf0100010 <entry+4>:   0x00000000      0x00000000      0x00000000
0x00000000
0xf0100020 <entry+20>:  0x00000000      0x00000000      0x00000000
0x00000000
0xf0100030 <relocated+1>:       0x00000000      0x00000000      0x00000000
0x00000000
```

继续往下执行一步，再查看两个地址往后16字

```
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/16xw 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
```

```
0x100020:       0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
0x100030:       0x00000000      0x110000bc      0x0056e8f0      0xfeeb0000
(gdb) x/16xw 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524ffe
0x7205c766
0xf0100010 <entry+4>:   0x34000004      0x0000b812      0x220f0011
0xc0200fd8
0xf0100020 <entry+20>:  0x0100010d      0xc0220f80      0x10002fb8
0xbde0fff0
0xf0100030 <relocated+1>:       0x00000000      0x110000bc      0x0056e8f0
0xfeeb0000
```

发现两个地址后16字的内容一样，说明地址映射完成。

> 新映射建立后的第一条指令是什么，如果映射配置错误，它还能不能正常工作?

把entry.S文件中的`%movl %eax, %cr0`注释，make clean，然后重新编译。在`jmp %eax`所在地址f010002a处加断点，运行到此处

```
(gdb) b *0Xf010002a
Breakpoint 1 at 0xf010002a: file kern/entry.S, line 68.
(gdb) c
Continuing.
Remote connection closed
(gdb) si
The program is not being run.
(gdb)
```

qemu也直接退出了，打印了错误信息

```
qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c

EAX=f010002c EBX=00010094 ECX=00000000 EDX=0000009d
ESI=00010094 EDI=00000000 EBP=00007bf8 ESP=00007bec
EIP=f010002c EFL=00000086 [--S--P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00007c4c 00000017
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00110000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00010094 CCD=80010011 CCO=LOGICL
```

```
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000
XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000
XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000
XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000
XMM07=00000000000000000000000000000000
GNUmakefile:163: recipe for target 'qemu-gdb' failed
make: *** [qemu-gdb] 已放弃 (core dumped)
```

练习8：

> 使用“%o”形式的模式打印

格式说明由“%”和格式字符组成，如%d%f等。它的作用是将输出的数据转换为指定的格式输出。格式说明总是由“%”字符开始的。

> 解释printf.c和console.c之间的接口。

printf.c中的putch函数调用了console.c中的cputchar函数 printf.c中的vcprintf函数调用了printfmt.c中的vprintfmt函数

> 从console.c解释以下内容：

```
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      }
```

查看console.h中的定义

```
#define CRT_ROWS        25
#define CRT_COLS        80
#define CRT_SIZE        (CRT_ROWS * CRT_COLS)
```

查阅得知CRT_COLS是显示器每行的字长，取值为80；CRT_ROWS是显示器的行数，取值为25；CRT_SIZE是显示器屏幕能够容纳的字数，取值为2000。 当crt_pos大于等于CRT_SIZE时，说明显示器屏幕已写满，因

此将屏幕的内容上移一行，即将第2行到最后1行的内容覆盖第1行到倒数第2行，然后将最后1行用黑色的空格塞满。最后更新crt_pos的值。

> 在调用cprintf（）时，fmt是什么意思？ ap是什么意思？

fmt指向字符串"x %d, y %x, z %d\n"的内存地址，ap指向x的内存地址。

> 列出（按执行顺序）以下每次调用cons_putc，va_arg和vcprintf这三段代码时的细节

```
cprintf调用vcprintf，调用时传入的第1个参数fmt的值为字符串"x %d, y %x, z %d\n"的地址，第2个参数ap指向x的地址。
vcprintf调用vprintfmt，vprintfmt函数中多次调用va_arg和putch，putch调用cputchar，而cputchar调用cons_putc，putch的第一个参数最终会传到cons_putc.接下来按代码执行顺序列出每次调用这些函数的状态。
第1,2次调用cons_putc: printfmt.c第95行，参数为字符'x'  ' '
第1次调用va_arg: printfmt.c第75行，lflag=0，调用前ap指向x，调用后ap指向y
第3次调用cons_putc: printfmt.c第49行，参数为字符'1'
第4,5,6,7次调用cons_putc: printfmt.c第95行，参数为字符',' ' ' 'y' ' '
第2次调用va_arg: printfmt.c第75行，lflag=0，调用前ap指向y，调用后ap指向z
第8次调用cons_putc: printfmt.c第49行，参数为字符'3'
第9,10,11,12次调用cons_putc: printfmt.c第95行，参数为字符',' ' ' 'z' ' '
第3次调用va_arg: printfmt.c第75行，lflag=0，调用前ap指向z，调用后ap指向z的地址加4s
第13次调用cons_putc: printfmt.c第49行，参数为字符'4'
第14次调用cons_putc: printfmt.c第95行，参数为字符'\n'
```

> 运行下面的代码 unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i); 输出是什么

输出为“He110 World”,10进制数57616转换到16进制结果为0xe110，在ASCII码中，0x00 0x64 0x6c 0x72对应值为'\0','d', 'l','r'，小端机器所以打印顺序位相反。

> 在下面的代码中，将在“y =”之后打印什么？ （注意：答案不是一个固定的值。）为什么会发生这种情况？ cprintf("x=%d y=%d", 3);

打印出来的y的值应该是栈中存储x的位置后面4字节代表的值。因为当打印出x的值后，va_arg函数的ap指针指向x的最后一个字节的下一个字节。

练习**9**：

> 确定内核在哪里完成了栈的初始化，以及栈所在内存的确切位置。内核如何为栈保留空间？栈指针初始化时指向的是保留区域的“哪一端”。

由前面的实验可知，当进入了entry.S文件也就意味着进入了操作系统内核。因此在entry.S文件中寻找内核对于栈的初始化。

```
relocated:

        # Clear the frame pointer register (EBP)
        # so that once we get into debugging C code,
        # stack backtraces will be terminated properly.
        movl    $0x0,%ebp                       # nuke frame pointer
```

```
        # Set the stack pointer
        movl    $(bootstacktop),%esp
```

结合注释可以看出这两句指令初始化了ebp,esp这两个栈指针。 查看kernel.asm代码可知栈顶位于 0xf0110000

```
        # Set the stack pointer
        movl    $(bootstacktop),%esp
f0100034:       bc 00 00 11 f0          mov    $0xf0110000,%esp
```

通过entry.S文件查看内核如何给栈保留空间

```
bootstack:
        .space          KSTKSIZE
```

在inc/memlayout.h中查询得知KSTKSIZE具体的大小

```
#define KSTKSIZE    (8*PGSIZE)          // size of a kernel stack
```

```
#define PGSIZE      4096        // bytes mapped by a page
```

由此可以计算出内核给栈分配了32KB的空间大小。 因为栈是由内存高位开始像低位扩展的，因此栈顶指针初始化指向的是高位。

练习**10**：

根据题目要求在obj/kern/kernel.asm中找到test_backtrace函数的地址

```
f0100040 <test_backtrace>:
```

可知test_backtrace函数的地址应为f01000401。 在kern/init.c文件中找到test_backtrace代码如下：

```
void
test_backtrace(int x)
{
        cprintf("entering test_backtrace %d\n", x);
        if (x > 0)
                test_backtrace(x-1);
        else
```

```
                mon_backtrace(0, 0, 0);
        cprintf("leaving test_backtrace %d\n", x);
}
```

根据题目要求在0xf0100040处设置断点，运行时qemu显示为：

```
6828 decimal is XXX octal!
```

最终qemu显示为：

```
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!
```

每次调用test_backtrace时，主要做了如下事情： 1.将返回地址（call指令的下一条指令的地址）压栈 2.将ebp, eip, ebx三个寄存器的值压栈，以便退出函数前恢复它们的值 3.调用cprintf函数打印"entering test_backtrace x"，x为输入参数的值 4.将参数(x-1)压栈 5.调用test_backtrace(x-1) 6.调用cprintf函数打印"leaving test_backtrace x"，x为输入参数的值

每次调用test_backtrace时共将8个双字压栈： 1.返回地址 2.ebp（记录当前函数栈帧的底部）, eip（记录下一条指令的地址）, ebx（记录当前的参数）这三个寄存器的值

练习**11**：

调用read_ebp函数来获取当前ebp寄存器的值。ebp寄存器的值实际上是一个地址，指向当前函数的栈帧的底部。因此ebp[1]存储的就是函数返回地址，ebp[2]以后存储的是输入参数的值。 entry.S中有初始化ebp的代码，如下：

```
relocated:

        # Clear the frame pointer register (EBP)
        # so that once we get into debugging C code,
        # stack backtraces will be terminated properly.
        movl    $0x0,%ebp                       # nuke frame pointer
```

因此递归结束的条件即为ebp等于初始值，也就是0的时候。

练习**12**：

> 在debuginfo_eip中__STAB_ *来自哪里?

STAB_BEGIN，**STAB_END**，**STABSTR_BEGIN**，__STABSTR_END__等都是在kern/kern.ld文件中定义，它们分别代表.stab和.stabstr这两个段开始与结束的地址。

```
        .stab : {
                PROVIDE(__STAB_BEGIN__ = .);
                *(.stab);
                PROVIDE(__STAB_END__ = .);
                BYTE(0)         /* Force the linker to allocate space
                                   for this section */
        }

        .stabstr : {
                PROVIDE(__STABSTR_BEGIN__ = .);
                *(.stabstr);
                PROVIDE(__STABSTR_END__ = .);
                BYTE(0)         /* Force the linker to allocate space
                                   for this section */
        }
```

```
objdump -h obj/kern/kernel
objdump -G obj/kern/kernel
gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL
-gstabs -c -S kern/init.c
```

> 看看bootloader是否在内存中加载了符号表，作为加载内核二进制文件的一部分

使用gdb查看符号表的位置是否存储有符号信息= =不太懂 查了半天stab找不到是什么东西，最后才发现是缩写。。原来就是符号表（symbol table）= = https://blog.csdn.net/cinmyheart/article/details/39972701 上面的链接中有简单介绍，看到stab结构体的定义，就能大概理解objdump -G obj/kern/kernel 输出的内容了。 当然在kdebug.c里的注释也有提到，也就看懂了个大概

```
// stab_binsearch(stabs, region_left, region_right, type, addr)
//
//      Some stab types are arranged in increasing order by instruction
//      address.  For example, N_FUN stabs (stab entries with n_type ==
//      N_FUN), which mark functions, and N_SO stabs, which mark source
files.
//
//      Given an instruction address, this function finds the single stab
//      entry of type 'type' that contains that address.
//
```

```
//      The search takes place within the range [*region_left,
*region_right].
//      Thus, to search an entire set of N stabs, you might do:
//
//              left = 0;
//              right = N - 1;     /* rightmost stab */
//              stab_binsearch(stabs, &left, &right, type, addr);
//
//      The search modifies *region_left and *region_right to bracket the
//      'addr'.  *region_left points to the matching stab that contains
//      'addr', and *region_right points just before the next stab.  If
//      *region_left > *region_right, then 'addr' is not contained in any
//      matching stab.
//
//      For example, given these N_SO stabs:
//              Index  Type   Address
//              0      SO     f0100000
//              13     SO     f0100040
//              117    SO     f0100176
//              118    SO     f0100178
//              555    SO     f0100652
//              556    SO     f0100654
//              657    SO     f0100849
//      this code:
//              left = 0, right = 657;
//              stab_binsearch(stabs, &left, &right, N_SO, 0xf0100184);
//      will exit setting left = 118, right = 554.
//
```

> 给内核模拟器增加backtrace命令

模仿kern/monitor.c中已有的命令即可

```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo", "Display information about the kernel", mon_kerninfo
},
        { "backtrace", "Display the backtrace of the function",
mon_backtrace },
};
```

补充

> 何处使用“%o”形式的模式打印八进制数字

才发现qemu一开始运行的时候就使用了八进制输出，将十进制的6828转换为了八进制为15254 `6828 decimal is 15254 octal!`