操作系统 Lab1 基础

宁晨然 17307130178

1.实验目的

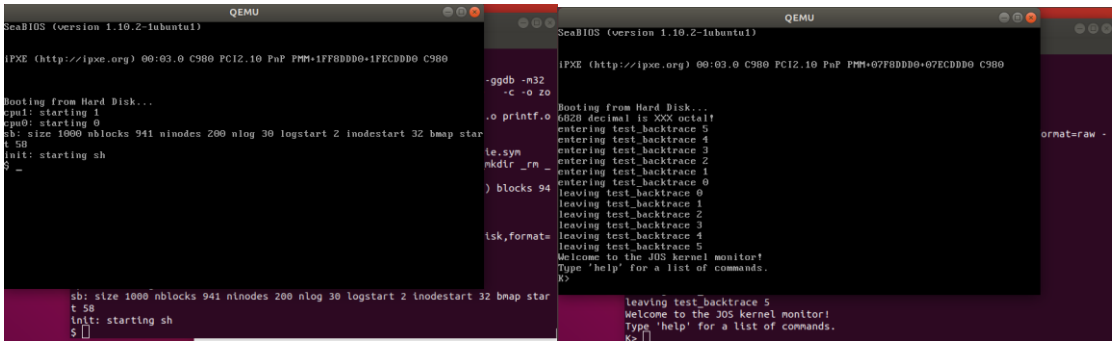
- 了解操作系统开发实验环境
- 熟悉命令行方式的编译、调试工程
- 掌握基于硬件模拟器的调试技术
- 了解 X86 汇编语言
- 了解 C 语言如何在 X86 机器上使用堆栈

2.实验过程

2.1 PC Bootstrap

根据实验教程配置好实验环境后，在 lab 中使用命令 make qemu-gdb 和 make gdb。利用 si 可以查看单步执行结果，利用 si N 查看多步执行结果。

```
# If the makefile can't find QEMU, specify its path here
QEMU = qemu-system-i386
```




```
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
```

2.2The Boot Loader

问题一：在地址 0x7c00 处设置断点，让程序继续运行到这个断点，跟踪 /boot/boot.S 文件的每一条指令，追踪到 bootmain()函数中，再具体追踪到 readsect()子函数，找出和 readsect() C 语言程序的每一条语句所对应的汇编指令，回到 bootmain()，然后找出把内核文件从磁盘读取到内存的那个 for 循环所对应的汇编语句。找到当循环结束后会执行哪条语句，在那里设置断点，继续运行到断点，然后执行完剩余的语句。

在 lab 中执行 make qemu-gdb 后可以运行 qemu 并且处于 stop 状态，打开另一个 terminal 执行 make gdb，可以开始调试 qemu。使用命令设置断点 b *0x7c00 后按 c，程序运行到 0x7c00 处并停止，此时 qemu 并没有完全 boot 初始化。

si 多次后比对/boot/boot.S 文件，可以找到对应的指令，执行完 boot.S 中的指令后，运行到 0x7d15 时，开始 bootmain()函数。由 boot.c 可知，bootmain()中调用了 readseg()，readseg()中调用了 readsect()函数。

①boot loader

boot loader 有两个主要的功能：它可以将处理器从实模式转换为 32bit 的保护模式，

从而可以访问超过 1MB 空间的内容。同时可以使用 x86 特定的 IO 指令直接访问磁盘设备寄存器，从磁盘中读取内核。而此处的 bootloader 由 boot.S 和 main.c 构成。BIOS 找到一个可以启动的软盘或者硬盘后，就会把这 512 字节的启动扇区加载到内存地址 0x7c00-0x7dff 区域内。所以在 0x7c00 设置断点，之后运行的就是 boot loader。

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

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/50 0x7c7c
   0x7c7c:      push   %ebp
   0x7c7d:      mov    %esp,%ebp
   0x7c7f:      push   %edi
   0x7c80:      mov    0xc(%ebp),%ecx
   0x7c83:      call   0x7c6a
   0x7c88:      mov    $0x1,%al
   0x7c8a:      mov    $0x1f2,%edx
   0x7c8f:      out    %al,(%dx)
   0x7c90:      mov    $0x1f3,%edx
   0x7c95:      mov    %cl,%al
   0x7c97:      out    %al,(%dx)
   0x7c98:      mov    %ecx,%eax
```

左图为设置断点后运行到了 0x7c00。而此时地址 0x7c7c 装载的就是 readsect()函数。

boot.S 的作用时，准备好从 16 位切换为 32 位工作模式的寄存器等工作，完成切换后，调用 main.c 中的 bootmain()。

②bootmain()

根据 si 继续执行，对比 main.c 的代码，可以看出 main.c 中各个函数的位置：

0x7d15	bootmain()
0x7cdc	readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
0x7c7c	readsect()

bootmain 函数的作用是加载启动扇区。

readseg 函数的作用是读取段。把距离内核起始地址 offset 个偏移量存储单元作为起始，并且把它和之后 count 字节的数据读出送入以 pa 为起始地址的内存物理地址处。此处的作用就是将内核第一个页（即 SECTSIZE*8）的内容读入到 ELFHDR 的内存地址处，即将操作系统映像文件的 elf 头读入内存。

readsect 函数的作用就是读取节。

ph 就是 program head table 表头，之后的 for 循环的作用就是，循环从外存读入到内存操作系统内核。

```
0x7d51:      cmp    %esi,%ebx
0x7d53:      jae    0x7d6b
0x7d55:      pushl  0x4(%ebx)
0x7d58:      pushl  0x14(%ebx)
0x7d5b:      add    $0x20,%ebx
0x7d5e:      pushl  -0x14(%ebx)
0x7d61:      call   0x7cdc
0x7d66:      add    $0xc,%esp
0x7d69:      jmp    0x7d51
0x7d6b:      call   *0x10018
```

左图为 bootmain 函数中的 for 循环和最后的一句话 0x7d6b

问题二： *处理器在什么时候开始执行 32 位程序？到底是什么引起了 16 位到 32 位模式的切换？*

在 boot.S 文件中可以看出有 16 位到 32 位的模式切换，切换的语句为 0x7c2d 处：

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

原因是因为 CPU 需要在保护模式下才能工作。

问题三： *boot loader 执行的最后一条指令是什么？它加载内核的第一条指令是什么？*

最后一条指令，是bootmain 函数的最后一条语句((void (*)(void)) (ELFHDR->e_entry))();

```
0x7d6b  call *0x10018
```

```
(gdb) b *0x7d6b
Breakpoint 2 at 0x7d6b
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d6b:      call   *0x10018

Breakpoint 2, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

该语句的意义就是跳转回操作系统内核程序的起始指令的地方。而内核被加载到内存之后执行的第一条指令就是 entry 的第一句话:

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

2.3 The Kernel

问题一：*内核在哪里初始化它的栈，栈在内存的什么地方？内核是怎样给栈保存空间的？栈初始指针是指向保留区域的哪一端？*

首先先追踪现在运行的指令。已知在 boot.S 调用 main.c 中的 bootmain()函数后，最后执行的语句是 call *0x10018，跳转到 kern/entry.S 中的 entry:位置。接下来执行的语句就是 movw $0x1234,0x472。由于现在还没有设置好虚拟内存地址，所以跳转 entry 的方式使用的是实地址，也是低地址位。

```
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb) info reg
eax            0x113600 1127936
ecx            0x0      0
edx            0xa4     164
ebx            0x10094  65684
esp            0x7bec   0x7bec
ebp            0x7bf8   0x7bf8
```

在bootmain()所有代码中没有修改过ebp和esp两个与栈息息相关的寄存器。说明在bootmain()里面没有初始化栈。浏览 entry.S 可知，最后一句执行的操作是调用 i386_init，说明在此处之前应该已经完成了栈的初始化。

观察到调用 i386_init()之前，对栈进行了修改，即栈的初始化。

```
# 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
```

esp 设置为 bootstacktop，ebp 清空为 0。对应的汇编代码如下。

```
0x10002f:    mov    $0x0,%ebp
0x100034:    mov    $0xf0110000,%esp
0x100039:    call   0x1000a6
```

在 0x1000c 设置断点后运行于此，打印出后面内存中存放的汇编代码，可得对应 call i386_init 语句前的两个栈初始化语句，ebp=0,esp=0xf0110000。

```
(gdb) si
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb) si
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
```

继续运行程序可以发现在 jmp *%eax 命令之后，地址从 0x10002d 变成了 0xf010002f，此时已经完成了从虚地址到实地址的映射，可以直接使用 0xf010002f 来表示实地址 0x10002f，也说明之前的几步操作是完成实虚地址转换。

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov     $relocated, %eax
jmp     *%eax
```

左图部分就是映射过程，根据注释可知，目前完成转换的部分是通过查表 entry_pgdir 的方式映射，这个手写的页表将虚拟地址[KERNBASE, KERNBASE+4MB)映射到实地址[0, 4MB)。对此实际情况即，[0xf0000000-0xf0400000]这 4MB 的虚拟地址空间映射为[0x00000000-0x00400000]的物理地址空间，此时的映射范围还比较局限。

说明 esp=0xf0110000 的地址并不是实际物理内存地址，而是虚拟地址。实际上赋值的值是 bootstacktop，在 bootstacktop 前定义了 bootstack，此时先开空间 KSTKSIZE，然后

定义 bootstacktop。而 KSTKSIZE=8*PGSIZE=8*4096=32KB，说明此处开空间的内容即用于堆栈，堆栈的大小就是 32KB。则实际用于堆栈的地址就是[0xf0108000-0xf0110000)，对应物理存储地址[0x00108000-0x00110000)

综上所述：

1. 内核在 entry.S 中 call i386_init 前初始化栈。
2. 栈在内存[0xf0108000-0xf0110000)的虚拟空间，即[0x00108000-0x00110000)物理地址空间。
3. entry.S 中的 bootstack 里面声明了一块 32KB 的空间用于堆栈。
4. 堆栈是向下生长，所以栈初始指针指向最高地址 bootstacktop，即 0xf0110000。

问题二： *修改程序，使程序能够正确输出%o（八进制数）*

首先找到与 printf()函数相关的文件，分别是 console.c/printf.c/printmt.c。先从 printf.c 几个函数入手。分析可知，putch,vcprintf,cprintf，后者都调用了前者，而 putch 最基本的函数就是 cputchar(),该函数在 console.c 中。

```
// `High'-level console I/O.  Used by readline and cprintf.

void
cputchar(int c)
{
        cons_putc(c);
}
```

cputchar()函数的作用是 console 调用的高层 IO 操作函数，被 readline 和 cprintf 使用。cons_putc()函数则是更基层的函数，作用是把一个字符输出到屏幕。所以 cputchar()的作用就是将字符输出到屏幕。

再观察关键的函数 cprintf()中调用了 vcprintf,调用了 printmt.c 中的 vprintmt()函数。由 lib/printmt.c 开头注释可知，这个文件是我们要修改的文件："打印各种样式的字符串的子程序，经常被 printf，sprintf，fprintf 函数所调用，这些代码是同时被内核和用户程序所使用的。"

找到 vprintmt()函数中 reswtich 部分，可以发现 u（十进制无符号数）的操作如下：

```
// unsigned decimal
case 'u':
        num = getuint(&ap, lflag);
        base = 10;
        goto number;
```

getuint()函数的作用是获取数字大小并分配变量。base 就是进制，此处是十进制，最后 goto number，进行数字操作。同理可以仿照写出八进制的内容。

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

仿照后面十六进制也可以知道，此处不需要额外添加 0，让用户自己添加即可。需要做的就是获取当前 num 和 base，然后传递给 number 继续做。结果正确！如下。

```
Booting from Hard Disk...
6828 decimal is 15254 octal!
```

问题三： *运行如下代码，观察并解释输出*

```
unsigned long long i = 0x5e2d5e6e61647546;
cprintf("H%x %s\n", 57616, &i);
```

如果要运行以上代码，需要加入到主函数当中。kern/monitor.c 就是当前运行的主函数所在位置，可以在 void monitor(struct Trapframe *tf)函数中看到 kernel 的 welcome 语句，所以在此处添加上以上代码后重新编译，得到结果：

```
Type 'help' for a list of commands.
He110 Fudan^-^\→▶
```

cprintf ("H%x %s\n",57616,&i);语句的意思就是输出，%x 即以十六进制形式输出 57616，而该数十六进制就是 e110，后面&i 是 int 类型 i 变量的地址，输出格式是%s 即在&i 位置处

的字符串，而 i 是一个 unsigned long long，其实就是将存放在 i 出的 int 类型当成 char 一个一个读取，并且此时是小端系统，读取后所得就是 Fudan^-^

0x5e(^) 2d(-) 5e(^) 6e(n) 61(a) 64(d) 75(u) 46(F)

问题四：*找到 obj/kern/kern.asm 中 test_backtrace 子程序的地址，设置断点，探讨在内核启动后，这个程序被调用时发生了什么？对于这个循环嵌套调用程序 test_backtrace，一共有多少信息压入到了堆栈之中？代表什么含义？*

首先打开 obj/kern/kern.asm，可以找到 test_backtrace 的子程序地址为 0xf0100040，如图。`f0100040 <test_backtrace>:` 设置断点。可发现 test_backtrace()在 kern/init.c 中。

```
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);
}
```

这个函数是一个递归调用自己的函数，作用是为了测试栈空间的运行是否正确。

```
// Test the stack backtrace function (lab 1 only)
test_backtrace(5);
```

在 i386_init() 函数中调用了 test_backtrace()，并且传递 5。

该程序进入是会打印 entering，出的时候会打印 leaving，而循环调用使得先进的后出。

为了了解栈帧的变化，我设置多个断点，考虑每次的 ebp 和 esp 的值。

① 进入 test_traceback()之前，找到 i386_init()汇编中的语句对应的断点位置为 0xf01000ef，运行到此时，ebp=0xf010fff8,esp=0xf010ffe0。说明 i386_init()函数的栈帧为[0xf010ffe0, 0xf010fff8)

```
0xf01000e3 <+61>:    call   0xf0100a49 <cprintf>
0xf01000e8 <+66>:    movl   $0x5,(%esp)
0xf01000ef <+73>:    call   0xf0100040 <test_backtrace>
```

②进入 test_traceback 前，先将返回地址压入栈中，此时 esp=0xf010ffdc，然后进入 test_traceback(5)。

首先 call 指令把 i386_init 的返回地址压入堆栈中，所以 esp 变为 0xf010ffdc，然后进入 test_backtrace(5)子程序。

```
(gdb) disassemble test_backtrace
Dump of assembler code for function test_backtrace:
=> 0xf0100040 <+0>:     push   %ebp
   0xf0100041 <+1>:     mov    %esp,%ebp
   0xf0100043 <+3>:     push   %esi
   0xf0100044 <+4>:     push   %ebx
   0xf0100045 <+5>:     call   0xf01001bc <__x86.get_pc_thunk.bx>
   0xf010004a <+10>:    add    $0x112be,%ebx
   0xf0100050 <+16>:    mov    0x8(%ebp),%esi
   0xf0100053 <+19>:    sub    $0x8,%esp
```

执行完左边的操作后才会执行 cprintf，此时可知

ebp = 0xf010ffd8

esp = 0xf010ffc8

之后循环调用时，同理：

test_backtrace(4)	ebp = 0xf010ffb8	esp = 0xf010ffa8
test_backtrace(3)	ebp = 0xf010ff98	esp = 0xf010ff88
test_backtrace(2)	ebp = 0xf010ff78	esp = 0xf010ff68
test_backtrace(1)	ebp = 0xf010ff58	esp = 0xf010ff38

对于每次压入栈中的内容有，push %ebp，把 i386_init 的 ebp 寄存器中的值压入栈中，即 esp 变为 0xf010ffd8。mov 后将 ebp 更新为 esp 的值，0xf010ffd8。push esi/ebx，所以传递了两个参数,esp=0xf010ffd0。之后的 sub 又使得 esp-8,此时是 test_traceback 为了分配给子程序 8 个存储单元的额外栈空间，用于存储一些临时变量。所以最终的 esp 为 0xf010ffc8。

压入栈中的就有 ebp/ebx/esi，其中 ebp 是返回地址，ebx 可能是子程序使用的寄存器

需要保存原有值，esi 是传递的参数 x，另外开了 8 个额外空间用于存储临时变量。一共有 16 个字的空间，每次压入栈中。

问题五：*实现栈回溯程序 mon_backtrace( ) (函数原型定义在 kern/monitor.c) -inc/x86.h 中的 read_ebp()函数*

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

首先为了添加命令，先在 command 总体命令中添加 backtrace 的 cmd。
然后在 mon_backtrace 中填充代码：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    unsigned int ebp, *ptr_ebp;
    ebp = read_ebp();   //get the ebp
    cprintf("Stack backtrace:\n");
    while(ebp!=0)
    {
        ptr_ebp = (unsigned int*)ebp;
        cprintf("\tebp = %x eip = %x\n arguments: %08x %08x %08x %08x %08x\n",
                ebp,ptr_ebp[1],ptr_ebp[2],ptr_ebp[3],ptr_ebp[4],ptr_ebp[5]);
        ebp = *ptr_ebp;
    }
    return 0;
}
```

可以使用 read_ebp()函数获取到 ebp 当前值，而 ebp 存放的值就是调用栈帧的 ebp，而 ebp+4 的地址存放的就是调用者的返回地址，再往上就是调用者调用的时候存放的传递参数，此处输出 4 个参数。下面是输出结果。

```
    ebp = f010ff18 eip = f0100078
arguments: 00000000 00000000 00000000 f010004a
    ebp = f010ff38 eip = f01000a1
arguments: 00000000 00000001 f010ff78 f010004a
    ebp = f010ff58 eip = f01000a1
arguments: 00000001 00000002 f010ff98 f010004a
    ebp = f010ff78 eip = f01000a1
arguments: 00000002 00000003 f010ffb8 f010004a
    ebp = f010ff98 eip = f01000a1
arguments: 00000003 00000004 00000000 f010004a
    ebp = f010ffb8 eip = f01000a1
arguments: 00000004 00000005 00000000 f010004a
    ebp = f010ffd8 eip = f01000f4
arguments: 00000005 00001aac 00000640 00000000
    ebp = f010fff8 eip = f010003e
arguments: 00000003 00001003 00002003 00003003
```

3.实验感受

本次实验从启动操作系统的角度出发，了解了计算机启动过程中遇到的三个步骤，BIOS 负责初始化总线及其他设备，并且搜索能启动的设备，如果发现了启动盘就读取盘内的 boot loader，并把控制权交给它。bootloader 负责将内存从实模式切换成保护模式，并且将 kernel 读入到内存中。kernel 负责开启内存分页，启动虚拟内存，实验 IO 操作，并且初始化栈。

实验中比较困难的一点就是需要结合汇编代码和源程序的阅读，了解程序运行的步骤，然后一一对应，设置断点去判断当前寄存器的值是否变化等。并且还加强了我对栈的理解，复习了一些以前计算机原理的基础知识。