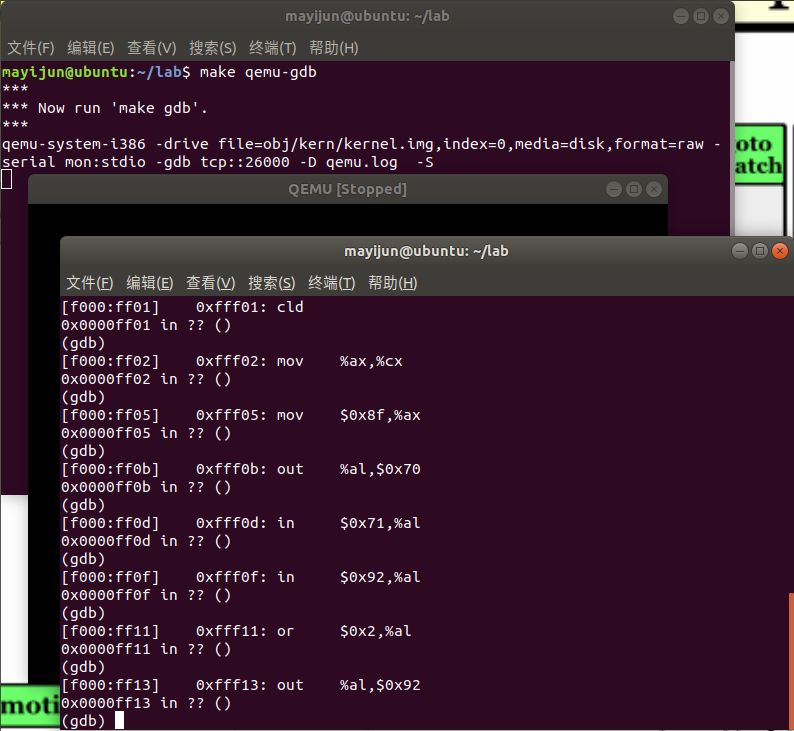
lab1 report

马逸君 17300180070

一、PC Bootstrap

配置实验环境xv6及JOS，然后利用GDB的si命令跟踪ROM BIOS的指令。

效果如图所示。

刚启动时，BIOS关闭了中断（cli），执行了多个in/out指令。查表知道，0x70和0x71端口用于控制CMOS，0x92则是PS/2。由此可见，BIOS正在执行自检，即确认基本设备正常连接正常工作。

二、The Boot Loader

Boot Loader工作过程综述：

BIOS先从模拟硬盘的第一扇区加载boot.S并在实模式下开始执行，即Boot Loader开始工作；

cli关闭中断，防止因跳转到其他代码执行、打断booting过程而出错；

初始化数据段、附加段、堆栈段的3个段寄存器为0；

打开A20 gate，使模拟计算机突破1MB寻址限制；

加载全局描述符表(global descriptor table, GDT)，为进入保护模式做准备；

通过改写cr0寄存器的PE位，从实模式切换到保护模式；

从16位切换到32位模式；

初始化保护模式下的5个段寄存器；

设置好栈指针%esp，然后从汇编语句中进入C函数，自此永不返回；

在C环境下，Boot Loader继续完成剩余的工作，读取硬盘的第一页并校验ELF头；

通过循环语句继续加载完成内核的剩余程序段，内核是从第二个扇区开始储存的；

然后执行内核，永不返回。

值得一提的是，在Makefrag文件（makefile在/boot下的一部分文件）中，有如下编译参数：

-N -e start -Ttext 0x7C00 -o

其中“-e start”指出了Boot Loader的入口地址为start，而“-Ttext 0x7C00”指出了代码段的起始地址为0x7c00，也就导致start位置的虚拟地址为0x7c00。

另外从/boot/main.c中的相关代码可以看出，内核被加载到的地址是0x100000。

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。

将gdb输出的汇编代码记录到文件并分析，得到答案。

以下每条C语句后跟着的汇编语句为与该条C语句对应的汇编指令。

void readsect(void \*dst, uint32\_t offset)

{

=> 0x7c7c: push %ebp

=> 0x7c7d: mov %esp,%ebp

=> 0x7c7f: push %edi

=> 0x7c80: mov 0xc(%ebp),%ecx

waitdisk();

=> 0x7c83: call 0x7c6a

outb(0x1F2, 1);

=> 0x7c88: mov $0x1,%al

=> 0x7c8a: mov $0x1f2,%edx

=> 0x7c8f: out %al,(%dx)

outb(0x1F3, offset);

=> 0x7c90: mov $0x1f3,%edx

=> 0x7c95: mov %cl,%al

=> 0x7c97: out %al,(%dx)

outb(0x1F4, offset >> 8);

=> 0x7c98: mov %ecx,%eax

=> 0x7c9a: mov $0x1f4,%edx

=> 0x7c9f: shr $0x8,%eax

=> 0x7ca2: out %al,(%dx)

outb(0x1F5, offset >> 16);

=> 0x7ca3: mov %ecx,%eax

=> 0x7ca5: mov $0x1f5,%edx

=> 0x7caa: shr $0x10,%eax

=> 0x7cad: out %al,(%dx)

outb(0x1F6, (offset >> 24) | 0xE0);

=> 0x7cae: mov %ecx,%eax

=> 0x7cb0: mov $0x1f6,%edx

=> 0x7cb5: shr $0x18,%eax

=> 0x7cb8: or $0xffffffe0,%eax

=> 0x7cbb: out %al,(%dx)

outb(0x1F7, 0x20);

=> 0x7cbc: mov $0x20,%al

=> 0x7cbe: mov $0x1f7,%edx

=> 0x7cc3: out %al,(%dx)

waitdisk();

=> 0x7cc4: call 0x7c6a

insl(0x1F0, dst, SECTSIZE/4);

=> 0x7cc9: mov 0x8(%ebp),%edi

=> 0x7ccc: mov $0x80,%ecx

=> 0x7cd1: mov $0x1f0,%edx

=> 0x7cd6: cld

=> 0x7cd7: repnz insl (%dx),%es:(%edi)

=> 0x7cd7: repnz insl (%dx),%es:(%edi)

... #repeats 129 times in total

=> 0x7cd9: pop %edi

=> 0x7cda: pop %ebp

=> 0x7cdb: ret

}

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。

ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

=> 0x7d3a: mov 0x1001c,%eax

=> 0x7d3f: movzwl 0x1002c,%esi

=> 0x7d46: lea 0x10000(%eax),%ebx

=> 0x7d4c: shl $0x5,%esi

=> 0x7d4f: add %ebx,%esi #最终%esi中是ph，%ebx中是eph

for (; ph < eph; ph++)

=> 0x7d51: cmp %esi,%ebx

**=> 0x7d53: jae 0x7d6b**

readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);

=> 0x7d55: pushl 0x4(%ebx)

=> 0x7d58: pushl 0x14(%ebx)

=> 0x7d5b: add $0x20,%ebx

=> 0x7d5e: pushl -0x14(%ebx)

=> 0x7d61: call 0x7cdc

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。

结合汇编语言的知识可以判断，上文中加粗的指令即为循环结束的条件跳转指令，其目标地址即为循环结束后将要执行的语句。

=> 0x7d6b: call \*0x10018

处理器从哪里开始执行32位程序？哪条指令引起了16位到32位模式的切换？

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

**ljmp $PROT\_MODE\_CSEG, $protcseg**

**.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

.code32是32位程序的开始。上文中加粗的指令引起16位到32位模式的切换。

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？

boot loader的最后一条指令是：（下方为其对应的C语句）

=> 0x7d6b: call \*0x10018

((void (\*)(void)) (ELFHDR->e\_entry))();

它所加载的内核的第一条指令是：

=> 0x10000c: movw $0x1234,0x472

三、The Kernel

内核初始化工作综述：

内核被链接到的地址是(KERNBASE + 1M)，这是因为通常的操作系统内核都要被链接到虚拟空间中高地址处的内核虚存区，低地址则留给用户编程用；

但Boot Loader将其加载到的地址是1M，因为Boot Loader工作时尚未启用虚拟内存，而物理内存大小有限，没有这么高的地址；

故内核执行之初需要建立一个页表来进行映射，所以我们的源代码在给内核预留的空间中的空闲地址建立了一个平凡的页表，将虚拟地址[KERNBASE, KERNBASE+4MB)映射到物理地址[0, 4MB)；

之后将页表起始处的物理地址写入cr3，并改写cr0的控制位，打开分页功能；

初始化栈，并将PC跳转至KERNBASE以上的有效虚拟地址，然后开始执行C代码，永不返回。

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

内核从这里开始初始化栈：

relocated:

movl $0x0,%ebp

程序中的汇编语句给%esp赋值为bootstacktop，这个标识符的定义是：

.data

.p2align PGSHIFT

.globl bootstack

bootstack:

.space KSTKSIZE

.globl bootstacktop

bootstacktop:

说明bootstacktop的地址是.data(数据段)+KSTKSIZE，且.data到.data+KSTKSIZE的空间预留。栈底指针%esp先指向最高地址处，后随着用户调用而逐渐降低。

由此可见，栈在内存中的位置为.data到.data+KSTKSIZE，由gdb调试可知，bootstacktop的地址为0xf0110000，计算得到栈的地址为0xf0108000-0xf0110000。

栈的初始指针指向保留区域的高地址方向的一端，内核给栈保留空间的方式是直接在entry.S的数据段中声明保留空间。

2.修改程序，使程序能够正确输出%o (八进制数)。

将/lib/printfmc.c中需要修改的语句替换为：

num = getuint(&ap, lflag);

base = 8;

goto number;

即可。

3.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);

输出结果为：

其中，e110是十进制数57616的十六进制表示，是输出字符串中%x的替换结果；

对于输出字符串中的%s，printf将i所在地址的内容按字符串解释并输出，因为一般的现代计算机均为小端存储方式，所以从i的存储地址起，内存中的内容依次为：0x46 0x75 0x64 0x61 0x6e 0x5e 0x2d 0x5e，按字符串输出即得到"Fudan^\_^"；

至于后面的一些乱码，是因为这段内容后面没有跟结束符'\0'，所以printf函数会继续向后读取内存中的内容并按字符输出，直至碰到0x0方停止。

4.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？

这个程序逐层递归调用自身，并在递归到最底层的时候调用栈回溯程序mon\_backtrace()打印栈帧信息，最后逐级返回。功能是测试栈和栈回溯程序是否正常工作。

至于压入堆栈的信息的大小，首先通过在各层中栈帧刚形成时（sub …, %esp语句执行完毕后）查询%ebp和%esp的值来得到当前层栈帧的大小。查询结果为：

|  |  |  |
| --- | --- | --- |
| 递归深度 | %ebp | %esp |
| test\_backtrace(5) | 0xf010ffd8 | 0xf010ffc8 |
| test\_backtrace(4) | 0xf010ffb8 | 0xf010ffa8 |
| test\_backtrace(3) | 0xf010ff98 | 0xf010ff88 |
| test\_backtrace(2) | 0xf010ff78 | 0xf010ff68 |
| test\_backtrace(1) | 0xf010ff58 | 0xf010ff48 |

由此可见，每层调用最初压入栈中的信息的大小都是0x1016 = 1610字节。另外在过程中会有其他信息压栈，会增加栈帧大小，增加的大小视具体代码而定（见下文）。

当调用到底层（即最终调用mon\_backtrace()）的时候，%esp的值为0xf010feb4，说明test\_backtrace()的各层一共向栈中压入了0xf010ffd8 - 0xf010feb4 = 0x12416 = 29210个字节的信息被压入了堆栈中。

对于这些信息的含义，分析test\_backtrace()中的push指令总共有两类。第一类是最初形成栈帧时：

void

test\_backtrace(int x)

{

f0100040: 55 push %ebp

f0100041: 89 e5 mov %esp,%ebp

f0100043: 56 push %esi

f0100044: 53 push %ebx

...

f0100053: 83 ec 08 sub $0x8,%esp

这些语句的含义是保存返回时的栈帧地址、生成当前调用的栈底、保存被调用者保存寄存器、生成当前调用的栈帧。堆栈内的信息是上一级调用的帧指针ebp地址（也是当前层返回时栈指针应该指向的地址。它的地址为：当前层ebp）、被调用者保存寄存器的内容（也就是上一层调用的“现场”，当前层调用返回时会恢复。地址依次位于上层ebp值的下方，此例中为ebp-8、ebp-4）、当前层的非静态局部变量（此例地址：[ebp-16, ebp-8） ）。

第二类是调用其他函数时：

f0100056: 56 push %esi

...

f010005d: 50 push %eax

f010005e: e8 e6 09 00 00 call f0100a49 <cprintf>

和

f010006d: 6a 00 push $0x0

f010006f: 6a 00 push $0x0

f0100071: 6a 00 push $0x0

f0100073: e8 0b 08 00 00 call f0100883 <mon\_backtrace>

这些push语句是为调用其他函数做准备，每次调用其他函数前压入堆栈的信息是调用者保存寄存器（地址依次位于参数上方）、调用其他函数的参数（从右向左依次压入栈中。地址：下一层ebp +8, +12, +16, …）和调用返回的地址（地址：下一层ebp +4）。

5. 实现栈回溯程序mon\_backtrace() (函数原型定义在kern/monitor.c)。（提示：使用inc/x86.h中的read\_ebp()函数）

（stab相关资料汇总："stab"是symbol table的缩写。stab是向调试器提供当前程序信息的一种格式。gcc编译程序时，若添加了-g参数，则会（对于通常的目标文件格式）以在生成代码中加入stab命令的方法将调试信息添加到目标文件中，这些命令将调试信息加到.o文件的符号表和字符串表中，并在链接时添加到可执行文件的符号表和字符串表中，供调试器使用。）

在/kern/kdebug.c中需要的位置添加如下代码：

stab\_binsearch(stabs, &lline, &rline, N\_SLINE, addr);

if (lline > rline) return -1;

info->eip\_line = stabs[lline].n\_desc;

然后实现mon\_backtrace()：

int mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)

{

uint32\_t \*cur\_ebp = (uint32\_t\*)read\_ebp(), \*prev\_ebp, \*caller\_eip;

struct Eipdebuginfo di;

cprintf("Stack backtrace:\n");

while (cur\_ebp != 0)

{

cprintf(" ebp %08x", cur\_ebp);

prev\_ebp = (uint32\_t\*)\*cur\_ebp; cur\_ebp++;

caller\_eip = (uint32\_t\*)\*cur\_ebp; cur\_ebp++;

cprintf(" eip %08x", caller\_eip);

cprintf(" args");

for (int i = 0; i < 5; i++) cprintf(" %08x", \*(cur\_ebp++));

cprintf("\n");

if (debuginfo\_eip((int)caller\_eip, &di) == 0)

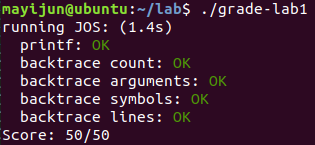
cprintf("\t\t%s:%d: %.\*s+%d\n", di.eip\_file, di.eip\_line, di.eip\_fn\_namelen, di.eip\_fn\_name, (uintptr\_t)caller\_eip - di.eip\_fn\_addr);

cur\_ebp = prev\_ebp;

}

return 0;

}

评分结果如示。

参考资料：

[1] [http://leenjewel.github.io/blog/2014/07/29/[(xue-xi-xv6)]-cong-shi-mo-shi-dao-bao-hu-mo-shi/](http://leenjewel.github.io/blog/2014/07/29/%5b(xue-xi-xv6)%5d-cong-shi-mo-shi-dao-bao-hu-mo-shi/)

【学习xv6】从实模式到保护模式 leenjewel 2014.7.29

[2] <https://www.cnblogs.com/wanghj-dz/p/3975107.html>

实地址模式与保护模式之间的切换 wanghj\_dz 2014.9.16

[3] <https://www.cnblogs.com/liubiyonge/p/9350494.html>

x86的控制寄存器CR0,CR1,CR2,CR3 liubiyongge 2018-7-22

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

XT, AT and PS/2 I/O port addresses Wim Osterholt 1994-6-11

[5] <http://sourceware.org/gdb/onlinedocs/stabs.html#Overview>

STABS Cygnus Support. Written by Julia Menapace, Jim Kingdon, David MacKenzie