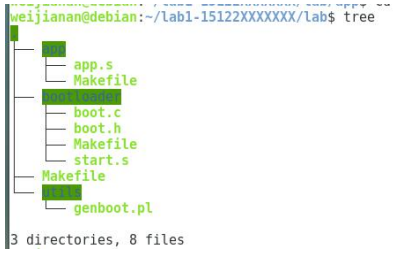
Lab1 系统引导实验报告

vigorweijia@foxmail.com

实验框架解读

实验环境

框架代码结构如下：



实验要求实现一个简单的引导程序，并在显示器上打印相应的内容。

按照 OS 网站上的教程，系统启动时，工作在实模式下，BIOS 会首先取得控制权，自检各个硬件是否正常工作，然后按 CMOS RAM 中设置的启动设备查找顺序来寻找可启动设备。工作在实模式的 BIOS 程序将主引导扇区（Master Boot Record，也就是磁盘 0 号扇区）加载至内存 0x7c00 处（被加载的程序一般称为 Bootloader），紧接着执行一条跳转指令，将 CS 设置为 0x0000，IP 设置为 0x7c00，运行被装入的 Bootloader。

```
1 os.img:
2         cd bootloader; make bootloader.bin
3         cd app; make app.bin
4         cat bootloader/bootloader.bin app/app.bin > os.img
5
6 clean:
7         cd bootloader; make clean
8         cd app; make clean
9         rm -f os.img
10
11 play:
12        qemu-system-i386 os.img
13
14 debug:
15        qemu-system-i386 -s -S os.img
```

需要执行这些指令，就必须有相应的硬件，本次实验使用的是硬件模拟软件 qemu。为了让 qemu 能够识别 x86 汇编代码，需要进行一系列的参数设定，如图，在根目录下的 Makefile 中 qemu 模拟的是 i386 架构，保证我们的实验能够顺利进行。

Makefile 文件解析

根目录下的 make 文件执行了 bootloader 中的 make 和 app 中的 make，然后强制把得到的 bootloader.bin 和 app.bin 拼接成 os.img，也就是我们实现的简单的系统内存镜像文件。由此，继续解读 bootloader 和 app 文件夹下的 Makefile 文件。

```
bootloader.bin: start.s boot.c boot.h
        gcc -c -m32 start.s -o start.o
        gcc -c -m32 -O1 -fno-stack-protector boot.c -o boot.o
        ld -m elf_i386 -e start -Ttext 0x7c00 start.o boot.o -o bootloader.elf
        objcopy -S -j .text -O binary bootloader.elf bootloader.bin
        ../utils/genboot.pl bootloader.bin

clean:
        rm -rf *.o *.elf *.bin
```

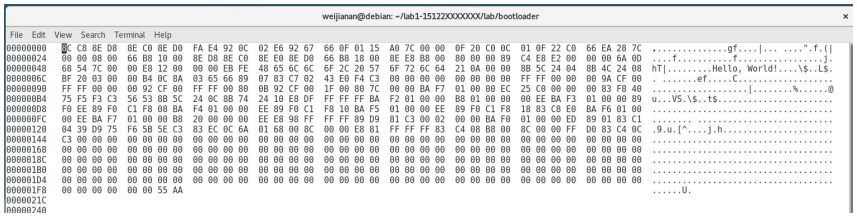
首先是编译 start.s 文件得到 start.o 文件，然后再编译 boot.c 得到 boot.o 文件，加-m32 参数是为了生成 32 位机器的汇编代码。在编译 boot.c 的时候使用 01 优化，并禁用栈保护措施，以免编译器优化掉一些代码。

接着是链接生成 bootloader.elf 文件。-m elf_i386 的意思是生成 i386 架构下的 elf 可执行文件。-e start 会解析”start”这个符号，并把它作为程序的入口地址。-Ttext 0x7c00 意思是把文件重定向到 0x7c00 处，这个可以通过 objdump 查看，如下：

```
weijianan@debian:~/lab1-15122XXXXXXX/lab/bootloader$ objdump -D bootloader.elf

bootloader.elf:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
    7c00:       8c c8                   mov    %cs,%eax
    7c02:       8e d8                   mov    %eax,%ds
    7c04:       8e c0                   mov    %eax,%es
    7c06:       8e d0                   mov    %eax,%ss
    7c08:       fa                      cli
    7c09:       e4 92                   in     $0x92,%al
    7c0b:       0c 02                   or     $0x2,%al
    7c0d:       e6 92                   out    %al,$0x92
    7c0f:       67 66 0f 01 15          lgdtw  (%di)
    7c14:       a0 7c 00 00 0f          mov    0xf00007c,%al
    7c19:       20 c0                   and    %al,%al
    7c1b:       0c 01                   or     $0x1,%al
    7c1d:       0f 22 c0                mov    %eax,%cr0
    7c20:       66 ea 28 7c 00 00       ljmpw  $0x0,$0x7c28
    7c26:       08 00                   or     %al,(%eax)
```

程序从 start 启动，start 放在内存位置的 0x7c00 处，也就是 IP 跳转后的位置，此时执行 bootloader 里面的内容。

接着是使用 objcopy 命令把 bootloader.elf 的内容拷贝到 bootloader.bin 中。由于无法对 bootloader.bin 使用 objdump 命令，使用 hexedit 工具查看它的二进制代码：



-S 参数是不拷贝重定位信息和符号信息。-j .text 意思是只拷贝.text 节。-O binary 意思是生成二进制文件。上面我通过 hexedit 查看 bootloader.bin 的二进制代码，通过与 bootloader.elf 对比，得知全为 bootloader.elf 的 text 段内容。（这里的 bin 文件已经经过了 perl 脚本处理）

最后对 bootloader.bin 执行了 utils 文件夹下的 genboot.pl 脚本。

```
#!/usr/bin/perl

open(SIG, $ARGV[0]) || die "open $ARGV[0]: $!";

$n = sysread(SIG, $buf, 1000);

if($n > 510){
        print STDERR "ERROR: boot block too large: $n bytes (max 510)\n";
        exit 1;
}

print STDERR "OK: boot block is $n bytes (max 510)\n";

$buf .= "\0" x (510-$n);
$buf .= "\x55\xAA";

open(SIG, ">$ARGV[0]") || die "open >$ARGV[0]: $!";
print SIG $buf;
close SIG;
```

首句是打开 bootloader.bin 这个文件，$!表示的是出错时的提示信息。然后读取 1000 个字节，最多读取 510 个字节（有 2 个预留给结束符），接着输出，不足 510 字节的用 0 填补。继续查阅资料，0x55 0xAA 是 MBR 的结束标志，执行完毕后 bootloader.bin 正好 512 字节，模拟了磁盘 0 号扇区的 512 字节。

然后查看 app 的 Makefile 文件。

```
app.bin: app.s
        gcc -c -m32 app.s -o app.o
        ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
        objcopy -S -j .text -O binary app.elf app.bin

clean:
        rm -rf *.o *.elf *.bin
```

如同 bootloader 的 Makefile 文件，会把 start 作为程序的入口，并把程序重定向到 0x8c00 这个位置，最后把 text 段拷贝到 app.bin 文件中。

框架代码解读

程序的入口是 start.s 中的 start，因此解读从 start.s 开始。

```
/* Real Mode Hello World */
.code16
#
.global start
#start:
#       movw %cs, %ax
#       movw %ax, %ds
#       movw %ax, %es
#       movw %ax, %ss
#       movw $0x7d00, %ax
#       movw %ax, %sp # setting stack pointer to 0x7d00
#       pushw $13 # pushing the size to print into stack
#       pushw $message # pushing the address of message into stack
#       callw displayStr # calling the display function
#loop:
#       jmp loop
#
#message:
#       .string "Hello, World!\n\0"
#
#displayStr:
#       pushw %bp
#       movw 4(%esp), %ax
#       movw %ax, %bp
#       movw 6(%esp), %cx #length of the string
#       movw $0x1301, %ax #ah = 0x13 print the string
#       movw $0x000c, %bx #bh = 0x00 black background bl = 0x0c red character
#       movw $0x0000, %dx #print ar line 0 col 0
#       int $0x10
#       popw %bp
#       ret
```

start 代码最开始处是被注释掉的实模式下输出 hello world。可以发现，在设定栈帧位置（0x7d00）后，就开始压入参数，执行打印字符串的函数 displayStr。int $0x10 实现的是 BIOS 中断，要求%ah = 0x13；%ax = 0x01 表示字符串只包含字符码，显示之后不更新光标位置，属性值在 BL 中。

接着是在保护模式下打印 Hello world。

```
/* Protected Mode Hello World */
.code16

.global start
start:
        movw %cs, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %ss
        cli # clear interuption
        inb $0x92, %al # Fast setup A20 Line with port 0x92, necessary or not?
        orb $0x02, %al
        outb %al, $0x92
        data32 addr32 lgdt gdtDesc # loading gdtr, data32, addr32
        movl %cr0, %eax
        orb $0x01, %al
        movl %eax, %cr0 # setting cr0
        data32 ljmp $0x08, $start32 # reload code segment selector and ljmp to start32, data32

.code32
start32:
        movw $0x10, %ax # setting data segment selector
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %fs
        movw %ax, %ss
        movw $0x18, %ax # setting graphics data segment selector
        movw %ax, %gs

        movl $0x8000, %eax # setting esp
        movl %eax, %esp
    # jmp to bootMain in boot.c
        calll bootMain
        pushl $13
        pushl $message
        calll displayStr
loop32:
        jmp loop32

message:
        .string "Hello, World!\n\0"

displayStr:
        movl 4(%esp), %ebx
        movl 8(%esp), %ecx
        movl $((80*5+0)*2), %edi #print hello world at line 5
        movb $0x0c, %ah #black background and red characters
nextChar:
        movb (%ebx), %al
        movw %ax, %gs:(%edi) # write graphics memory
        addl $2, %edi # %edi is the address of the character
        incl %ebx
        loopnz nextChar # loopnz decrease ecx by 1
        ret

.p2align 2
gdt: # 8 bytes for each table entry, at least 1 entry
        .word 0,0 # empty entry
        .byte 0,0,0,0

        .word 0xffff,0 # code segment entry
        .byte 0,0x9a,0xcf,0

        .word 0xffff,0 # data segment entry
        .byte 0,0x92,0xcf,0

        .word 0xffff,0x8000 # graphics segment entry
        .byte 0x0b,0x92,0xcf,0

gdtDesc: # 6 bytes in total
        .word (gdtDesc - gdt -1) # size of the table, 2 bytes, 65536-1 bytes, 8192 entries
        .long gdt # offset, i.e. linear address of the table itself
```

首先使用 cli 指令关中断，启动 A20 总线，查阅资料得知，8086 模式下，A20 打开的情

况下，访问超过 1MB 内存，就真实的访问。然后加载 GDTR，设置 CR0 的 PE 位为 1，表示打开分段。通过长跳转设置 CS 进入保护模式，初始化 DS，ES，FS，GS，SS 这几个段寄存器。这里设置了三个 GDT 表项，其中代码段与数据段的基地址都为 0x0，视频段的基地址为 0xb8000。

这之后，把参数压栈，转而执行 displayStr 函数输出字符串。%ebx 保存了字符串的首地址；%ecx 保存了需要打印的长度；%edi 保存了打印的位置，一行有 80 个字符，将在第 5 行第 0 列打印 hello world，查阅资料得知一个字符占用两个字节的显存，其原因是还需要一个字节保存一些属性，即背景色和字符颜色；%ah 保存背景色和字符颜色。接着就是在 nextChar 这个循环中输出字符串。ret 之后进入 loop32 这个死循环，否则会继续执行下面的代码，导致出错。

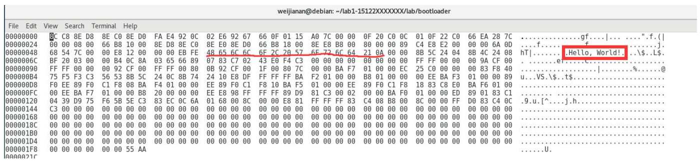
神奇的是，字符串的内容是直接保存在代码中的，readelf 找不到数据区中相应的内容，这可以直接从 objdump 或者 hexedit 工具读出。

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00007c00 000c00 000145 00  AX  0   0  4
  [ 2] .eh_frame         PROGBITS        00007d48 000d48 000070 00   A  0   0  4
  [ 3] .comment          PROGBITS        00000000 000db8 00002d 01  MS  0   0  1
  [ 4] .symtab           SYMTAB          00000000 000de8 000140 10      5  13  4
  [ 5] .strtab           STRTAB          00000000 000f28 00007a 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000fa2 000034 00      0   0  1
```

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00007000 0x00007000 0x00db8 0x00db8 R E 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```

```
00007c54 <message>:
    7c54:       48                      dec    %eax
    7c55:       65 6c                   gs insb (%dx),%es:(%edi)
    7c57:       6c                      insb   (%dx),%es:(%edi)
    7c58:       6f                      outsl  %ds:(%esi),(%dx)
    7c59:       2c 20                   sub    $0x20,%al
    7c5b:       57                      push   %edi
    7c5c:       6f                      outsl  %ds:(%esi),(%dx)
    7c5d:       72 6c                   jb     7ccb <readSect+0x13>
    7c5f:       64 21 0a                and    %ecx,%fs:(%edx)
      ...
```

字符串的ASCII码



即符号表、只读数据区找不到存储的字符串。

直接执行这段代码，就可以在保护模式下输出 hello world。

由于中断关闭，无法通过陷入磁盘中断调用 BIOS 进行磁盘读取，本次实验通过 readSec(void *dst, int offset)这一接口读写（in，out 指令）磁盘的相应端口（Port）来实现磁盘特定扇区的读取。

```
#include "boot.h"

#define SECTSIZE 512

void bootMain(void) {
        void (*elf)(void);
        elf = (void(*)(void))0x8c00;
        readSect((void*)elf, 1); // loading sector 1 to 0x8c00
        elf(); // jumping to the loaded program
}

void waitDisk(void) { // waiting for disk
        while((inByte(0x1F7) & 0xC0) != 0x40);
}

void readSect(void *dst, int offset) { // reading a sector of disk
        int i;
        waitDisk();
        outByte(0x1F2, 1);
        outByte(0x1F3, offset);
        outByte(0x1F4, offset >> 8);
        outByte(0x1F5, offset >> 16);
        outByte(0x1F6, (offset >> 24) | 0xE0);
        outByte(0x1F7, 0x20);

        waitDisk();
        for (i = 0; i < SECTSIZE / 4; i ++) {
                ((int *)dst)[i] = inLong(0x1F0);
        }
}
```

跳转到0x8c00

readSec 里面有许多看起来奇怪的向指定端口写入的指令。

查阅资料得知，这是 LBA28 方式读取磁盘，使用 28 个比特位来表示逻辑区号（每 512 字节为 1 个扇区），一共有 8 个端口，编号为 0x1f0~0x1f7。读入的长度放入 1f2（读入的扇区个数），1f3~1f6 保存的是逻辑区号，因此 offset 可以理解为需要读入的起始的磁盘扇区号。0x1f0 是硬盘接口的数据端口，并且是 16 位的，0x1F1 是端口的错误寄存器，里面保留

着“硬盘驱动器”最后一次执行工作的时候状态。（错误原因）

0x1F7是命令端口，又是状态端口。
发送0x20，硬盘里面的“管理员”就开始做一些准备工作了，并且把会0x1F7里面的第7个位置变成1代表自己很忙。
0010 0000 =我要拿东西
1010 0000 =我开始准备了
0010 1000 =我准备好了

waitDisk 等待磁盘空闲，此时就通过 inLong 函数读入磁盘内容并写到 dst 里，也就是内存起始地址为 0x8c00 的位置，最后 app.s 的内容从磁盘被读入内存。

在 app 的 Makefile 文件中，用户程序会被写入地址为 0x8c00 的位置，因此 elf()会跳转到用户程序中。

由此，框架分析完毕，我们需要在 start.s 中跳转到 bootMain，并在 app.s 中写入输出 hello world 的代码。

代码编写

当不需要返回时，可以直接在 start.s 中写入 jmp bootMain，为了保持原结构，我写的是 call bootMain。

```
.code32
start32:
        movw $0x10, %ax # setting data segment selector
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %fs
        movw %ax, %ss
        movw $0x18, %ax # setting graphics data segment selector
        movw %ax, %gs

        movl $0x8000, %eax # setting esp
        movl %eax, %esp
# jmp to bootMain in boot.c
        calll bootMain
        pushl $13
        pushl $message
        calll displayStr
```

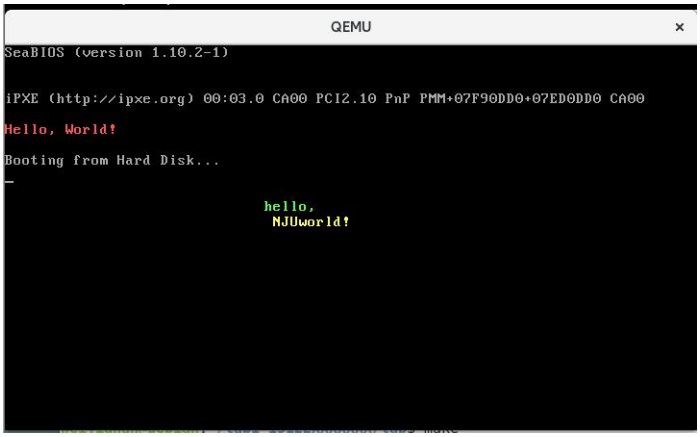
在 app.s 中，可以仿照 start.s 中保护模式的代码打印 hello world。不过为了维护栈帧的完整，并且在执行完后能够返回 bootMain 进而返回 start 中，需要创建一个新的栈帧，结构为: push %ebp; mov %esp,%ebp; ... ; leave; ret;

由于已经知道在输出时各个寄存器的作用，可以把代码改写得复杂一点，即在两行输出，并且输出不同的颜色。

```
1  .code32
2
3  .global start
4  start:
5          # TODO
6          push %ebp         #Create new stack frame
7          mov %esp, %ebp
8          pushl $16         #length of the string
9          pushl $message    #address of the string
10         calll displayStr
11         leave
12         ret               #Recover stack
13 message:
14         .string "hello, NJUworld!\n\0"
15 displayStr:
16         movl 4(%esp), %ebx
17         movl 8(%esp), %ecx
18         subl $0xa, %ecx #print 6 characters
19         movl $0xa, %edx
20         movl $((80*10+30)*2), %edi #print at line 10, col 30
21         movb $0x0a, %ah #color setting
22 nextChar1:
23         movb (%ebx), %al
24         movw %ax, %gs:(%edi)
25         addl $2, %edi
26         incl %ebx
27         loopnz nextChar1
28         movl %edx, %ecx #print 10 characters
29         movl $((80*11+30)*2), %edi
30         movb $0x0e, %ah #set new color
31 nextChar2:
32         movb (%ebx), %al
33         movw %ax, %gs:(%edi)
34         addl $2, %edi
35         incl %ebx
36         loopnz nextChar2
37         ret
```

每次执行 loopnz 时，%cx = %cx-1，如果%cx != 0 并且 zf = 0，会跳转到后面的标签，执行循环。

执行结果如下：



红色的 hello world 是保护模式下的输出，绿色和黄色的 hello world 是从磁盘加载的程序的输出。

心得体会

刚开始进行实验的时候比较茫然，甚至不理解要干什么，从哪里入手。仔细阅读了助教的 ppt、网站上的实验教程以及代码框架后，对整个程序的执行流程有了一个比较清晰的认识，然后才写出了代码，这也验证了助教说的读代码两小时，写代码 5 分钟。

但是，对于本次实验，还是有疑问，首先是显存的问题，每个字符占用两字节，有一个字节保存了颜色，但是同时%ah 寄存器了保存了颜色信息，二者感觉有点重复。