Report for Lab05-加载 OS 进入 main

Stu：金泽文

No.PB15111604

实验目的：

学习并掌握 makefile，bios 13h。学会加载操作系统映像。学习并掌握堆栈的空间分布与初始化栈的方法。学习并掌握汇编中调用 c 函数的方法，与 c 文件中调用汇编函数的方法。加深对操作系统启动过程的理解。

实验内容：

1. 学习制作简单的 makefile
2. 学习制作一个简单的操作系统映像
3. 学习从启动代码转入操作系统代码运行
4. 为 C 函数的运行做好适当的准备

相关原理学习：

1.Makefile 的学习。

① Why makefile ?

i. Makefile 避免了重复的冗余操作，尤其是切换主机和控制台时。

ii. Makefile 避免了修改单个文件时需要编译其他文件的多余开销。

② Makefile 的规则：

```
target ... : prerequisites ...
        command
        ...
```

```
    ...
```

i. Makefile 最核心的规则：如果某一个 prerequisites 比它对应的 target 对应的文件要新，那么 target 对应的文件需要重新执行对应的 command。

ii. 无参数的 make 将执行 makefile 第一个 target 对应的 command，根据依赖的 `prerequisites` 递归查找。

iii. 每条执行命令之前都要有一个 tab

iv. 可以通过=来定义宏，并通过$()的方式调用。

v. .PHONY target 表示的是为目标文件，作为标签使用。

vi. ‘\’ 表示换行符。

vii. 注释符为“#”。

viii. 一个 Makefile 中可以 include 其他文件

ix. 这样也是可以的。

```
target ... : prerequisites ...;command
  command
  ...
```

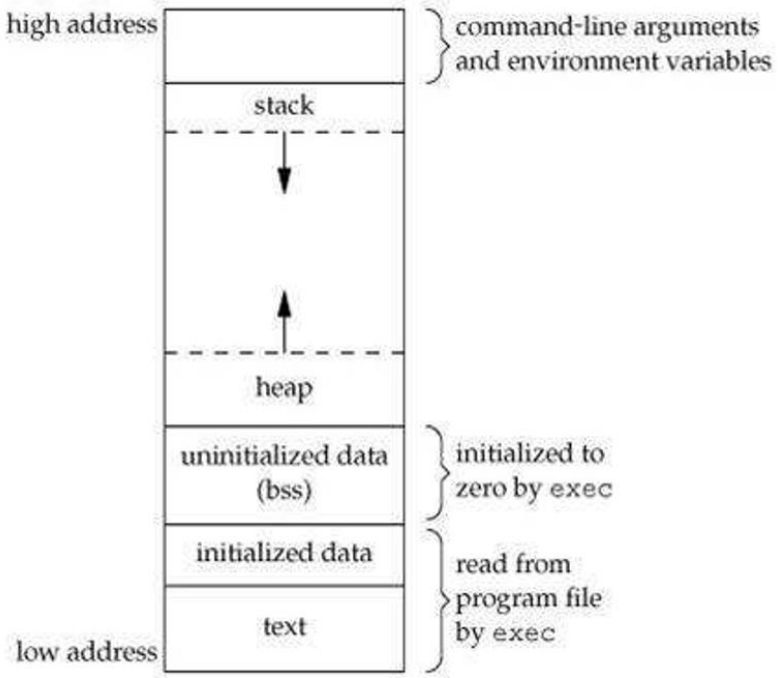
x. 通配符支持 * ~ ？，同 bash/shell 是一样的。

Makefile 的学习暂时到这里为止。将重点放在实验上。

2.bios INT 13h

中断号	寄存器		作用
13h	ah=00h	dl=驱动器号（0表示A盘）	复位软驱
	ah=02h ch=柱面（磁道）号 dh=磁头号 es:bx→数据缓冲区	al=要读扇区数 cl=起始扇区号 dl=驱动器号（0表示A盘）	从磁盘将数据读入es:bx指向的缓冲区中

3.linux 内存中进程布局。



栈由高地址向低地址增长。

实验过程：

1. start16.s：

   i. 从软盘加载操作系统映像到 0x7e00 开始的内存中。

加载之前需要重置 7c00 开始的 MBR，按照【相关原理学习】中第 2 点，重置需要的是 int 13h， ah=00h， dl=00h.

重置之后，需要利用 bios13 读取 floppy 并加载到内存中。

根据助教给的提示，扇区数设为 7，磁头号和驱动器号均设为 0，磁道号为 0，开始扇区为 2。结合【相关原理学习】中第 2 点，得到以下代码：

```
86      # 开始加载到内存
87      movw $0, %dx
88      movb $0, %ch
89      movb $2, %cl
90      movw %cs, %ax
91      movw %ax, %es
92      movw $0x7e00, %bx
93      movb $2, %ah
94      movb $7, %al
95      int $0x13
```

ii． 跳转到 start32.s

由于 start32.s 对应部分会加载到 0x7e00，所以只需要跳转到 0x7e00 即可。可以由一句

```
164 jump_to_32:
165     jmp $0x10, $0x7e00
```

实现。（我实现的代码中 0x10 是 code section descriptor）。

2. start32.s：

i． 栈的初始化

根据 ld 中的_end 变量，以及栈由低地址向高地址增长的原理，我们在初始化栈时，只需要给栈指针 esp 和帧指针 ebp 赋值为_end 加上栈的长度得到的结果。而一般我们将栈长度设置为 4k，所以可以得到一下代码：

```
4 _start32:
5       movl $_end, %eax
6       addl $0x1000, %eax
7       movl %eax, %esp
8       movl %eax, %ebp
```

ii． bss 的清空

我们只需要在 bss 处赋 0,即可。而 bss 首地址尾地址由 ld 脚本给出，为

_bss_start 和_bss_end。得到代码如下：

```
10  init_bss:
11      movl $_bss_end, %eax
12      movl %eax, %ebx
        # %ebx = _bss_start
13      subl $_bss_start, %eax
        # %eax = length of bss
14      movl $0, %ecx
        # %ecx = counter
15  bss_loop:
16      movb $0, 0(%ebx, %ecx, 1
            )
17      incl %ecx
18      cmpl %eax, %ecx
19      jz bss_loop
20
```

iii． 调用 C 的 main 中函数

直接 call main 里的函数即可，得到

```
23  call_main:
24      call fun
```

3. C 文件

i． 清屏：

直接在 vga 对应地址，即 0xb8000 处开始写 0 即可。

```
5       int i;
6       char *vga = (char *)0xb8000;
7       for(i = 0; i < 600; i++)
8           *(vga + i) = 0;
```

ii． 输出姓名学号：

直接输出。

```
    for(i = 0; i < 18; i++){
    *(vga + i*2) = string[i];
    *(vga + i*2 + 1 ) = 0x2f;
    }
```

4. 编写 makefile

I. 根据编译的依赖关系，得到以下文件。实现的额外功能：make clean

```
all: OS.bin 16.bin

OS.bin: OS.elf
    objcopy -O binary OS.elf OS.bin

16.bin: 16.elf
    objcopy -O binary 16.elf 16.bin

OS.elf: 32.o main.o 32.ld
    ld -T 32.ld 32.o main.o -o OS.elf -g

16.elf: 16.o 16.ld
    ld -T 16.ld 16.o -o 16.elf -g

16.o: 16.s
    gcc -c  16.s -o 16.o  -m32 -g

32.o: 32.s
    gcc -c  32.s -o 32.o  -m32 -g

main.o: main.c
    gcc -c main.c -o main.o -g -std=c99 -m32

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

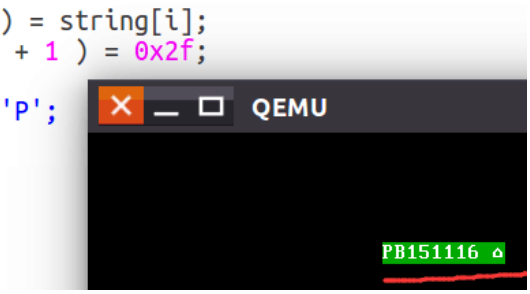
5. 运行与调试：

I. 为了方便调试，编写了两个脚本，一个模拟用，一个调试用。

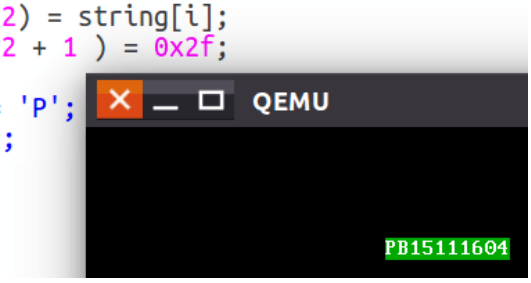
II. 调试过程中发现**有趣**的事情：

在最后 main.c 文件中输出字符串时，出现以下情况，

```
char string[] = "PB15111604";
int fun(){

    int i;

    for(i = 0; i < 600; i++)
        *((char *)0xb8000 + i) = 0;


    for(i = 0; i < 10; i++){
        *((char *)0xb8350 + i*2) = string[i];
        *((char *)0xb8350 + i*2 + 1 ) = 0x2f;
    }
    //*((char *)0xb8400 + 0) = 'P';
    //*((char *)0xb8402) = 'B';

    for(i = 0; i < 100; i++)
        i--;
    return 0;
}
```



```
char string[] = "PB15111604mdzzzzzzzzzzz";
int fun(){

    int i;

    for(i = 0; i < 600; i++)
        *((char *)0xb8000 + i) = 0;


    for(i = 0; i < 10; i++){
        *((char *)0xb8350 + i*2) = string[i];
        *((char *)0xb8350 + i*2 + 1 ) = 0x2f;
    }
    //*((char *)0xb8400 + 0) = 'P';
    //*((char *)0xb8402) = 'B';

    for(i = 0; i < 100; i++)
        i--;
    return 0;
```



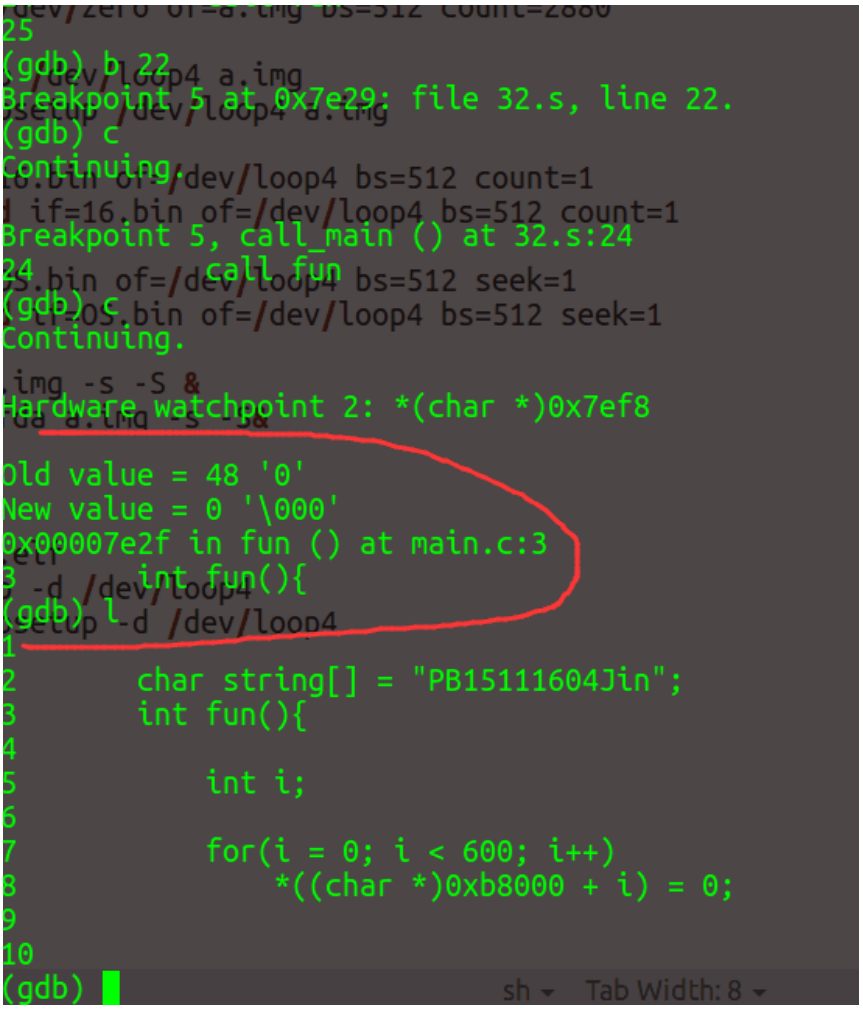
也就是：字符串个数影响了输出是否正常。

通过 hexdump -c a.img 得到

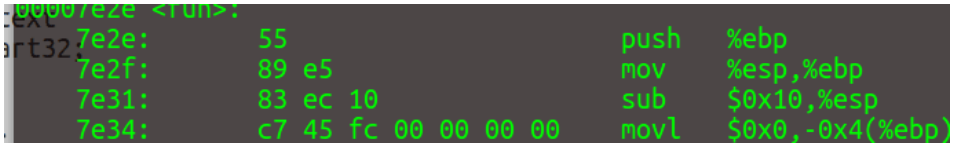
```
00002e0 005 002   |   @ \f 004 004  \0  \0  \0  \0  \0  \0  \0  \0  \0
00002f0   P   B   1   5   1   1   1   6   0   4   m   d   z   z   z  \0
0000300  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
```

发现，对齐的时候无乱码，差一点没有对齐的时候出现了乱码，而 img 本身是没有问题的。所以猜测是运行的时候“栈”的某些操作导致了这一结果的发生。而认真分析了代

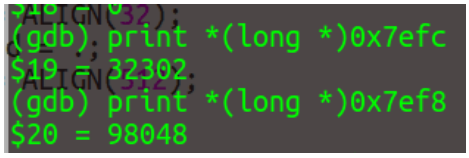
码，并没有发现任何异常，而在通过 gdb 找到字符串地址，并且一步步调试之后定位到问题发生的地方：



通过 wa *(char *)0x7ef8 定位到这里。发现是在



刚进入 fun 函数时的 push %ebp 处*(char *)0x7ef8 的值，由‘0’变成了‘\0’。但是查看 ebp 和 esp 发现二者都在栈顶离 0x17f00，相差甚远。不知道为何会改变 0x7ef8 的值。继续调试，查看所改写的新内容，发现：



对应的 16 进制表示为：

f8 对应的是 00 01 7f 00 fc 对应的是 00 00 7e 2e

通过 grep 查看反汇编代码，得到：

```
→ os objdump -d OS.elf |grep 7f
    7e00:       b8 00 7f 00 00          mov    $0x7f00,%eax
    7e0e:       a1 00 7f 00 00          mov    0x7f00,%eax
    7e15:       2b 05 00 7f 00 00       sub    0x7f00,%eax
→ os objdump -d 16.elf |grep 7f
    7c7f:       8c c8                   mov    %cs,%eax
    7d7f:       20 61 72                and    %ah,0x72(%ecx)
→ os objdump -d OS.elf |grep 7e2e
    7e29:       e8 00 00 00 00          call   7e2e <fun>
00007e2e <fun>:
    7e2e:       55                      push   %ebp
→ os ./qemu.sh
```

7f00 是_end 对应的地址，7e2e 是 fun 的入口，而 00 01 7f 00 是我的栈顶地址。

之后我把 ld 中的 16 位对齐改成了 32 位对齐，发现字符串附近没有改写。

同时，将栈长度从 0x10000 改成了 0x1000 之后发现 16 位对齐的情况写也不会出现改写的情况。我的猜测是：由于 16 位对齐的情况下，0x10000 的栈长度太长，所以需要堆内存储一下 ebp 和函数入口。就此，此案告破。（可是，为什么会覆盖掉原本的数据呢？）

6. 实验结果与截图：

1. make：

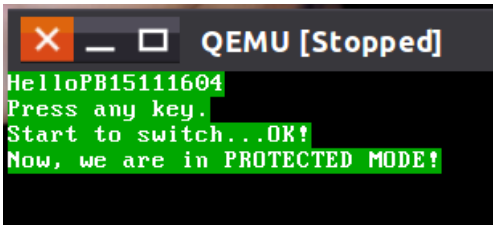
```
→ os make
gcc -c  32.s -o 32.o  -m32 -g
gcc -c main.c -o main.o -g -std=c99 -m32
ld -T 32.ld 32.o main.o -o OS.elf -g
objcopy -O binary OS.elf OS.bin
gcc -c  16.s -o 16.o  -m32 -g
ld -T 16.ld 16.o -o 16.elf -g
objcopy -O binary 16.elf 16.bin
cat all.sh >all
chmod a+x all
→ os
```

2. 运行脚本 qemu.sh

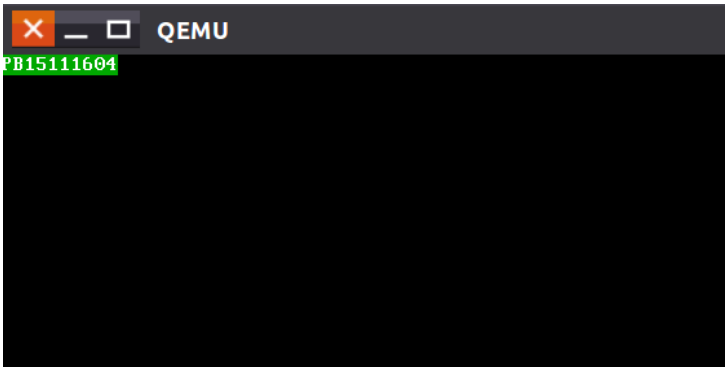
```
chmod a+x all
➜  os ./qemu.sh
make: 'all' is up to date.
make
记录了2880+0 的读入
记录了2880+0 的写出
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.0602789 s, 24.5 MB/s
dd if=/dev/zero of=a.img bs=512 count=2880

[sudo] zevin 的密码：
sudo losetup /dev/loop4 a.img

记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.00302325 s, 169 kB/s
sudo dd if=16.bin of=/dev/loop4 bs=512 count=1

记录了0+1 的读入
记录了0+1 的写出
267 bytes copied, 0.0101491 s, 26.3 kB/s
sudo dd if=OS.bin of=/dev/loop4 bs=512 seek=1


qemu -fda a.img
sudo losetup -d /dev/loop4

➜  os WARNING: Image format was not specified for 'a.img' and probing g
         Automatically detecting the format is dangerous for raw images
rations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions
```

3. 实模式

4. 进入保护模式（调试状态下的截图）



5. 调用 c 函数，清屏输出



要求的内容：

1. make 工具的作用是什么？在命令行上输入“make OS”的含义是什么？Makefile 中规则的语法是什么？

   作用在【相关原理学习】第一点中有详细说明。；make OS 的含义是在当前目录下找到 GNUmakefile、makefile 或 Makefile 文件，找到其中以 OS 为 target 的一行，按照对应的依赖链，依次执行 command，最终编译得到 OS 镜像。

   Makefile 中的语法在【相关原理学习】第一点中有详细说明。

2. 何时加载操作系统映像合适？加载多少个扇区合适

   由于需要利用 bios 13h，所以在进入保护模式之前加载比较合适，加载 7 个扇区。

3. 如何利用 BIOS 软盘驱动加载操作系统映像？

   这一点在【实验过程】第 1 点的第 i 点有详细说明。

4. 从汇编进入 C 需要做哪些准备？栈在什么位置比较合适？什么是 BSS 段？BSS 段清 0 有什么好处

   清空 bss 段，初始化栈。栈底从 bss 结束开始出比较合适，栈顶在栈底+0x1000 处比较合适。（栈长度如果比较大，如 0x10000，则会像我在【实验过程】中第 5 点所遇到的那样会出现一些额外的情况。）BSS 段是存放未初始化的变量的地方，清 0 的好处就是我们在未初始化的情况下使用时不会出现奇怪的数字。

5. 使用 C 语言编写写 VGA 缓存和汇编写 VGA 缓存有什么不一样？附加问题：你能不能从 C 语言中调用汇编写的 VGA 输出函数？你能不能从汇编调用 C 写的 VGA 输出函数？说出你的方法。

   ①其实在我看来 c 语言和汇编写 vga 本质是一样的，不同之处可能就在于汇编有 bios int 可以用，写起来麻烦；而 c 语言其实有很多库可以用，也可以直接赋值给*(char *)0xb8000。②要想从 C 语言调用汇编写的 vga 函数，只需要一个 fun.s（如右图）

```
#fun.s
.type fun, @function
.globl fun
fun:
    pushl %ebp
    mov %esp, %ebp
    #body
    leave
    ret
```

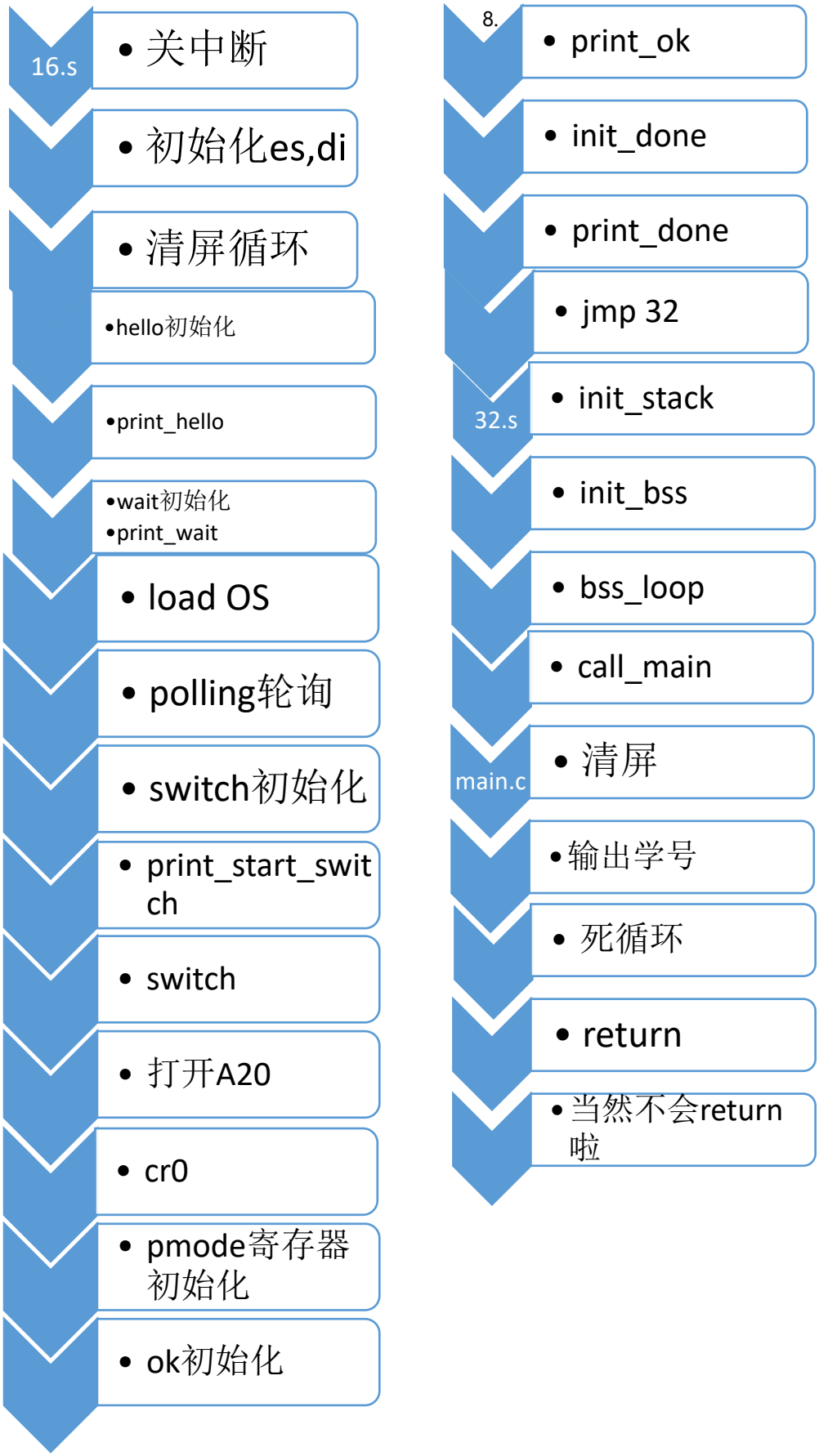
   并且在对应的 c 文件里声明：

   int fun(int, int); 并且一起编译即可。③在汇编调用 c 写的 vga 函数只需要像本次实验中 call main 一样，并且和 main.c 一起编译即可。不再赘述。

6. 给出代码运行关键时刻的截屏并加以说明

   这一点在【实验过程】中有详细的解答。

7. 给出你所有代码的流程图

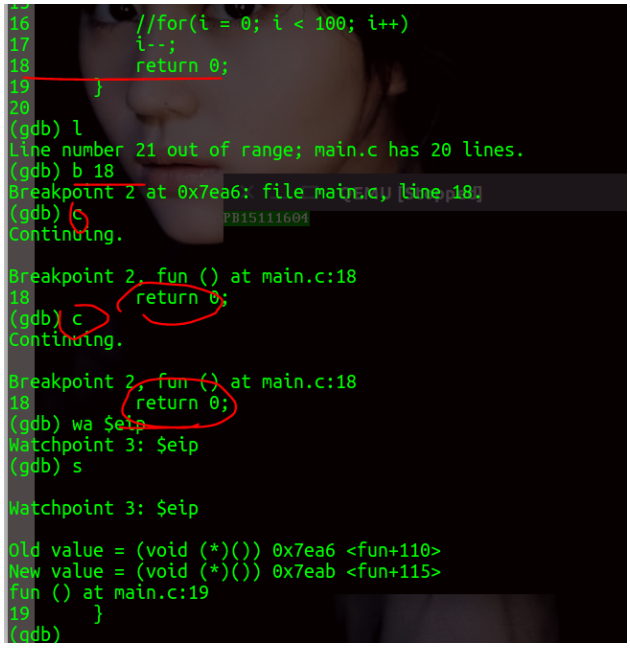

16.s
• 关中断
• 初始化es,di
• 清屏循环
•hello初始化
•print_hello
•wait初始化
•print_wait
• load OS
• polling轮询
• switch初始化
• print_start_switch
• switch
• 打开A20
• cr0
• pmode寄存器初始化
• ok初始化
8.
• print_ok
• init_done
• print_done
• jmp 32
32.s
• init_stack
• init_bss
• bss_loop
• call_main
main.c
• 清屏
•输出学号
• 死循环
• return
• 当然不会return啦

8. 说明你打包提交的源代码中的文件清单，要说清每个文件的作用

16.s	实模式汇编代码
32.s	保护模式汇编代码
main.c	包含 fun 函数的 c 文件
16.ld	16.s 对应 ld 脚本
32.ld	32.s 以及 main.c 对应 ld 脚本
Makefile	Makefile（包含 clean 功能）
qemu.sh	Shell 脚本

9. 附加问题：如果你的 main 函数最后不是死循环，请说明 main 函数返回后你的操作系统在执行什么？

首先，我猜测 main 函数返回到 call main 下一条指令，之后会继续执行。我猜会有一件有趣的事情发生。之前在调试的时候，我发现 main 函数和 32.s 指令在地址空间是挨着的，所以其实 call main 之后就是实际的 main 函数指令，所以还会 push%ebp mov %esp,%ebp，并且继续清屏，输出，右边是检验



这一过程的 gdb 截图：可以看到 return 0 运行了两次。

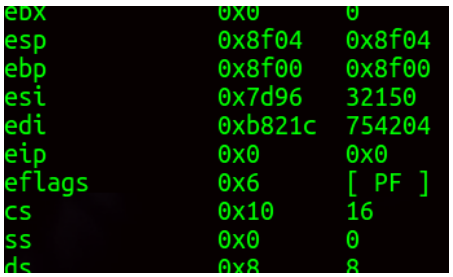
但是需要注意的是：最后的 leave, ret，这一次对应的 call 并不是在 32.s。所以我不太确定会 ret 到哪里。Google 一下之后，发现：

CPU执行ret指令时，进行下面的两步操作：

（1）(IP) = ((ss)*16 +(sp))

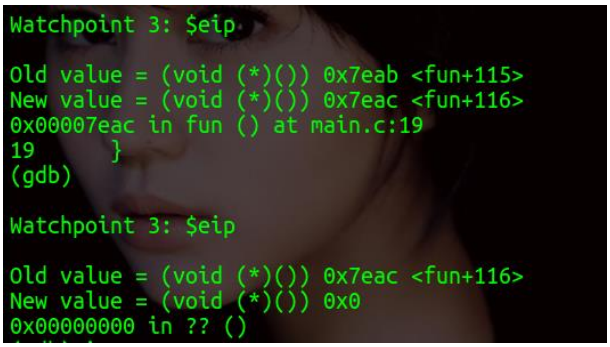
（2）(sp) = (sp)+2

通过 i r 查看 ss：



但是实际跟踪之后却发现：

eip 从 0x7eac 变成了 0x0！



继续跟踪，发现：

eip 变成了 0xe05b，而这正是 bios 的进入地址！于是可以预想，将会发生跟之前完全一样的启动过程，一直循环，直到从外部 kill qemu。