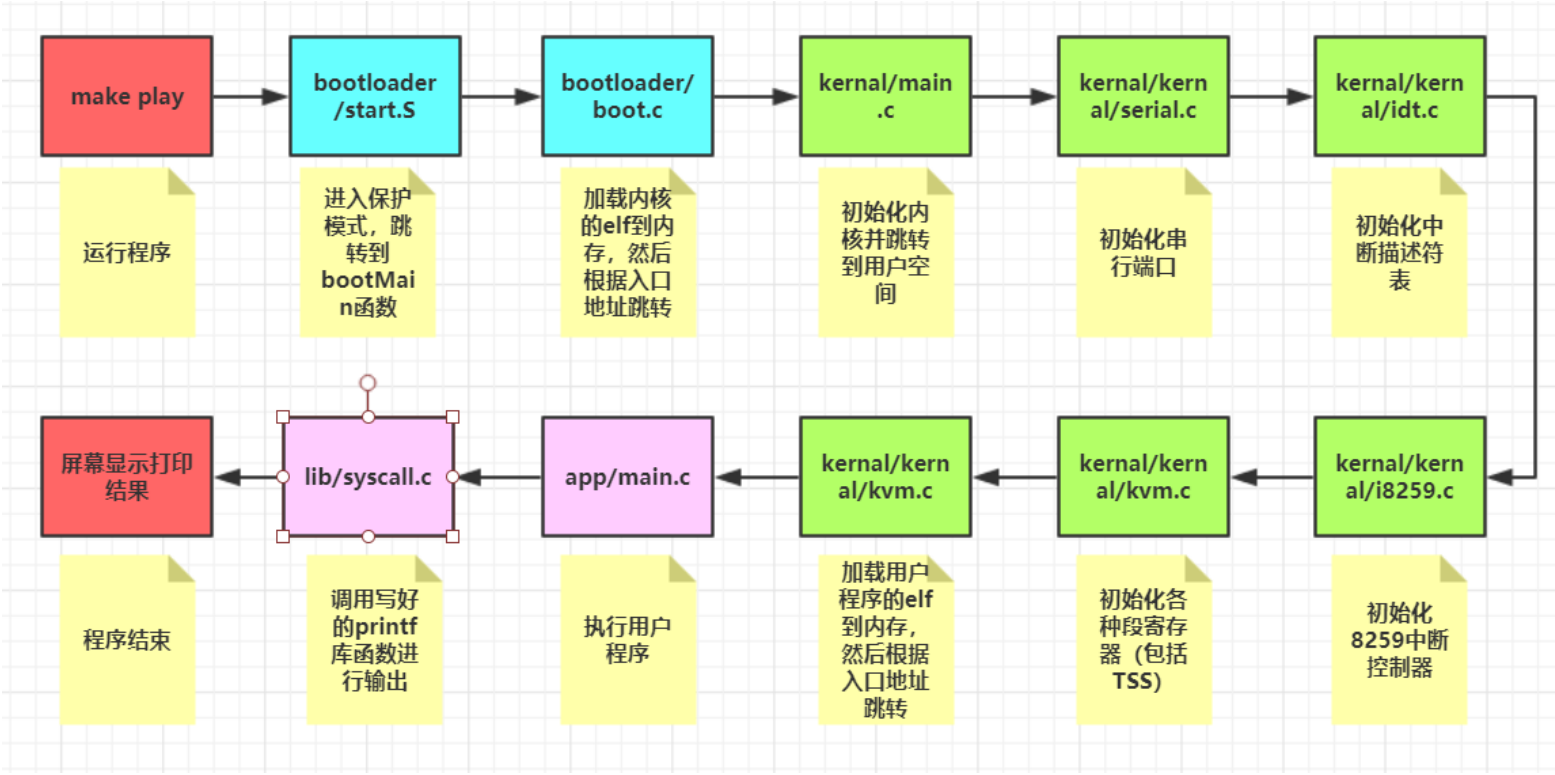
lab2 实验报告

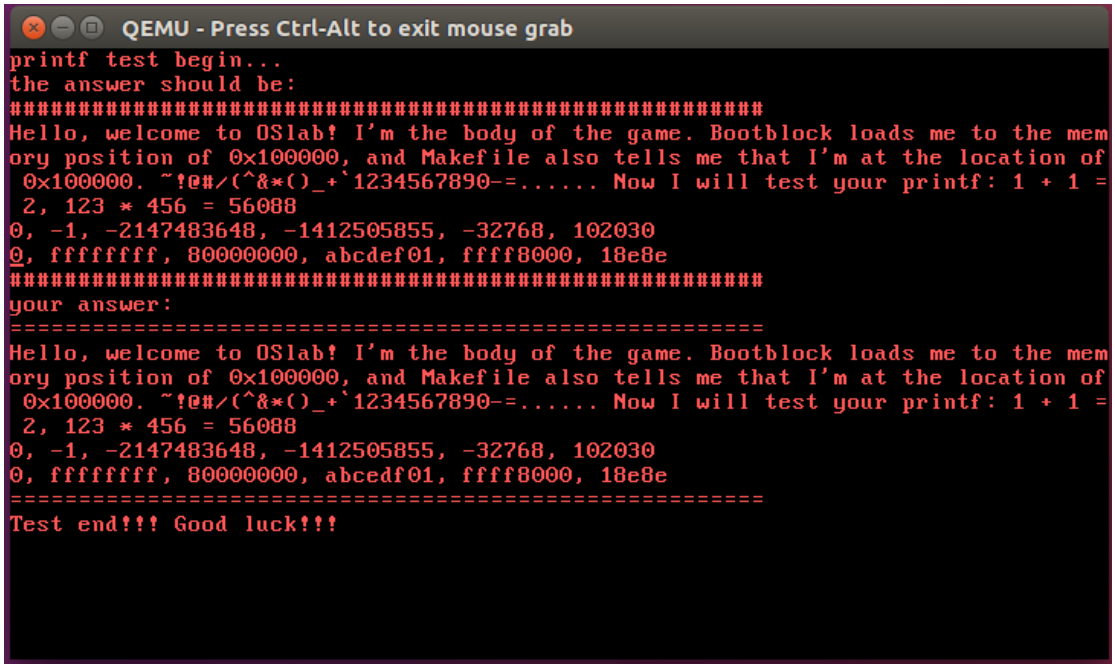
161220124 王明

1.1 代码流程与实验结果

代码流程图解：



运行结果如下：



1.2.实现内核代码和用户代码的装载

实模式向保护模式的切换：

在 start.S 的 code16 中完成关闭中断、开启 A20 总线、加载 GDTR，将 cr0 寄存器的 PE 位由 0 变成 1、再用 ljmp 指令跳转至保护模式这些操作完成，这部分代码模仿 lab1，代码如下：

```
.global start
start:

        cli
        inb $0x92, %al
        orb $0x02, %al
        outb %al, $0x92
        data32 addr32 lgdt gdtDesc
        movl %cr0, %eax
        orb $0x01, %al
        movl %eax, %cr0
        data32 ljmp $0x08, $start32
```

加载内核：

将 1 到 200 扇区的内容读到一块内存然后根据 elf 头和程序头表加载内核代码到内存（具体过程下面的代码有注释），然后通过 entry 这个入口地址去执行内核代码

```
/* 加载内核至内存，并跳转执行 */
char *p = (char *)0x8000;    //set loading addr
int i;
for(i = 1; i <= 200; i++)      //load sector 1 to 200
        readSect(p + (i - 1) * 512, i);
struct ELFHeader *elf_header;
struct ProgramHeader *ph_beg, *ph_end;
elf_header = (struct ELFHeader *)p;  //get elf_header
ph_beg = (struct ProgramHeader *)(p + elf_header->phoff);
ph_end = ph_beg + elf_header->phnum;  //get first ph & last ph for elf_header
struct ProgramHeader *pt;
for(pt = ph_beg; pt < ph_end; pt++)   //load according to ph's contents
{
        int j;
        for(j = 0; j < pt->memsz; j++)
        {
                if(j < pt->filesz)
                        *(char *)(pt->paddr + j) = *(char *)(p + pt->off + j);
                else
                        *(char *)(pt->paddr + j) = 0;
        }
}
void (*entry)(void);
entry = (void *)elf_header->entry;  //get entry_addr from elf_header
entry();   //jump to next program
```

初始化中断描述符表：

模仿示例代码添加 0 到 12 号中断门，其实都没用到，都是 assert(0)，代码如下：

```
 * init your idt here
 * 初始化 IDT 表，为中断设置中断处理函数
 */
setTrap(idt + 0x0, SEG_KCODE, (uint32_t)irq0, DPL_KERN);     //add 13 Traps
setTrap(idt + 0x1, SEG_KCODE, (uint32_t)irq1, DPL_KERN);
setTrap(idt + 0x2, SEG_KCODE, (uint32_t)irq2, DPL_KERN);
setTrap(idt + 0x3, SEG_KCODE, (uint32_t)irq3, DPL_KERN);
setTrap(idt + 0x4, SEG_KCODE, (uint32_t)irq4, DPL_KERN);
```

实现系统调用：

系统实现了 TrapFrame 这一框架，write 是 4 号调用接口，代码如下

```
void syscallHandle(struct TrapFrame *tf) {
        /* 实现系统调用*/
        switch(tf->eax)
        {
                case 4:  //sys_write matches number 4
                        tf->eax = print_on_screen((char *)tf->ecx, tf->edx); //ecx record addr of st
                        break;
                default:
                        assert(0);
        }
}
```

这个调用了 print_on_screen 函数，这个是自己实现的，比较简单，就是通过写显存的方式将字符串输出到屏幕上，为了输出方便，自己写了一个清屏函数 clear_screen(),代码如下：

```
void clear_screen()  //to clear the screen
{
        short *addr = (short *)0xB8000;
        int i;
        for(i = 0; i < screen_width * screen_height; i++)
        {
                *(addr + i) = 0;
        }
        cur = 0;
}
```

0xB8000 是 GS 的首地址，也就是显存地址。

加载用户程序：

这一部分代码和上面加载内核的代码几乎就是一样，那么就不再贴代码了。

这部分加载用户程序的代码是在内核执行的，那么下面要跳转到用户空间去执行用户程序，必须使用 iret 指令返回，恢复用户空间的各种通用寄存器和段寄存器，代码如下：

```
 */
asm volatile("sti");                                    // open interrupt
asm volatile("pushl %%eax"::"a"(USEL(SEG_UDATA)));      //push u_ss
asm volatile("movl $0x800000, %eax");                   //push esp
asm volatile("pushl %eax");
asm volatile("pushfl");                                 //push eflags
asm volatile("pushl %%eax"::"a"(USEL(SEG_UCODE)));      //push u_cs
asm volatile("pushl %0"::"r"(entry));                   // push eip

asm volatile("movw %%ax, %%es":: "a" (KSEL(SEG_UDATA))); //set u_es
asm volatile("movw %%ax, %%ds":: "a" (KSEL(SEG_UDATA))); //set u_ds

asm volatile("iret"); //pop  eip & u_cs & eflags & esp & u_ss
```

这部分使用了内联汇编的方式，具体的含义已经注释在后面。

1.3. 完善 printf 库函数

实现 printf 的格式化输出：

其实本质上就是解析字符串，将符合格式化输出的参数进行解析，最终保存在一个字符串数组里，我写了四个子函数去分别处理%d %x %c %s 格式的输入，具体的过程很简单，就不在此贴代码了，写的子函数如下：

```
 8 void process_s(char *s);     //process string
 9 void process_c(char c);      //process char
10 void process_x(uint32_t x); //process unsigned integer
11 void process_d(uint32_t d); //process signed integer
```

进行系统调用：

根据示例代码完善 syscall 函数，采用内联汇编进行将参数传递给寄存器，代码如下：

```
14 int32_t syscall(int num, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
15 {
16         int32_t ret = 0;
17
18         /* 内嵌汇编 保存 num, a1, a2, a3, a4, a5 至通用寄存器*/
19
20         asm volatile("int $0x80": //num for eax, a1 for ebx, a2 for ecx, a3 for edx
21                         "=a"(ret):
22                         "a"(num), "b"(a1), "c"(a2), "d"(a3), "D"(a4), "S"(a5));
23
24         return ret;
25 }
```

因为 printf 是用的系统函数中的 write 函数，因此自己写了个 sys_write 去调用 syscall，其实本质上就是将系统调用号设置为 4，将需要打印的字符串首地址赋给%ecx，将字符串的长度赋给%edx，从而完成 write()的参数传递，代码如下：

```
1 int32_t sys_write(int fd, char *buf, int len)
2 {
3         return syscall(4, (uint32_t)fd, (uint32_t)buf, len, 0, 0); //4 means write
4 }
```

1.4. 实验感想

这次实验花了很多时间学了很多课上没提的知识，真心有点累，还是希望以后的实验能够讲得更详细些吧，当然也有可能是我太菜了（笑哭）。