

南京大学本科生实验报告

课程名称：操作系统

任课教师：叶保留

助教：尹熙喆/水兵

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	201220062	姓名	黄子睿
Email	201220062@smail.nju.edu.cn	开始/完成日期	3/8-3/9

1. 实验名称：

OS LAB 2

2. 实验目的：

在第一次实验中，我们熟悉了操作系统启动的过程，本次实验将继续完善我们的 miniOS。我们知道，大部分的计算任务的过程：**接受输入->处理->输出**。要让他能够正常工作，我们需要使 OS 具备基本的**输入输出能力**。

3. 实验内容：

1.内容索引：

a) 回答了全部 exercise、challenge 与 conclusion

[Exercise 1](#)

[Exercise 2](#)

[Exercise 3](#)

[Exercise 4](#)

[Exercise 5](#)

[Exercise 6](#)

[Exercise 7](#)

[Exercise 8](#)

[Exercise 9](#)

[Exercise 10](#)

[Exercise 11](#)

[Exercise 12](#)

[Exercise 13](#)

[Exercise 14](#)

[Exercise 15](#)

[Exercise 16](#)

[Exercise 17](#)

[Exercise 18](#)

[Exercise 19](#)

[Exercise 20](#)

[Exercise 21](#)

[Exercise 22](#)

[Exercise 23](#)

[Challenge 1](#)

[Challenge 2](#)

[Challenge 3](#)

[Conclusion 1](#) [Conclusion 2](#) [Conclusion 3](#)

b) 实现了 scanf

c) 对 irqHandle.c, serial.c, abort.c 中的个别函数做了修改

2.具体回答:

Exercise 1 既然说确定磁头更快 (电信号), 那么为什么不把连续的信息存在同一柱面号同一扇区号的连续的盘面上呢? (Hint: 别忘了在读取的过程中盘面是转动的)

答: 因为在读取扇区的时候磁盘会转动, 加入把连续的信息存在同一柱面号同一扇区号的连续的盘面上, 虽然定磁头不需要时间, 但是磁盘转动复位的时间时需要考虑的。而假如先在同一盘面的磁道上存, 然后下一盘面的磁道上按顺序存, 磁盘的转动是同一方向的, 所需的时间比来回改变磁盘转动方向要少, 效率更高。

Exercise2: 假设 CHS 表示法中柱面号是 C, 磁头号是 H, 扇区号是 S; 那么请求一下对应 LBA 表示法的编号 (块地址) 是多少 (注意: 柱面号, 磁头号都是从 0 开始的, 扇区号是从 1 开始的)。

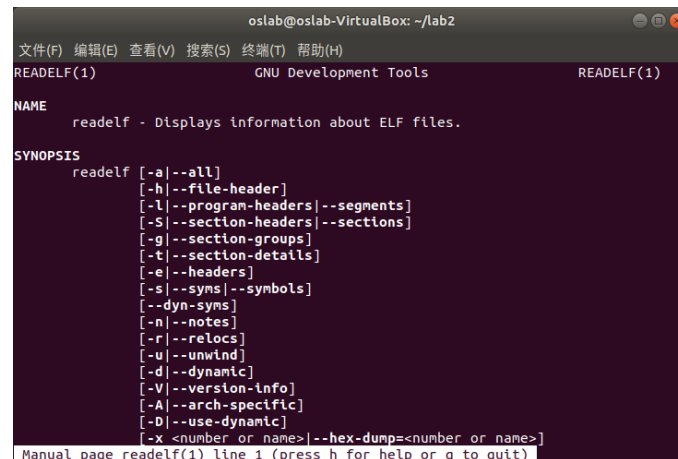
答: 不妨记柱面共 N1 个 (编号 0 - N1-1), 磁头共 N2 个 (编号 0 - N2-1), 扇区共 N3 个 (编号 1 - N3)。

则 CHS 表示法下记录为(C, H, S)的扇区在 LBA 表示法下可以表述为:

$$\text{Num} = \text{CN1} + \text{HN2} + \text{S} - 1$$

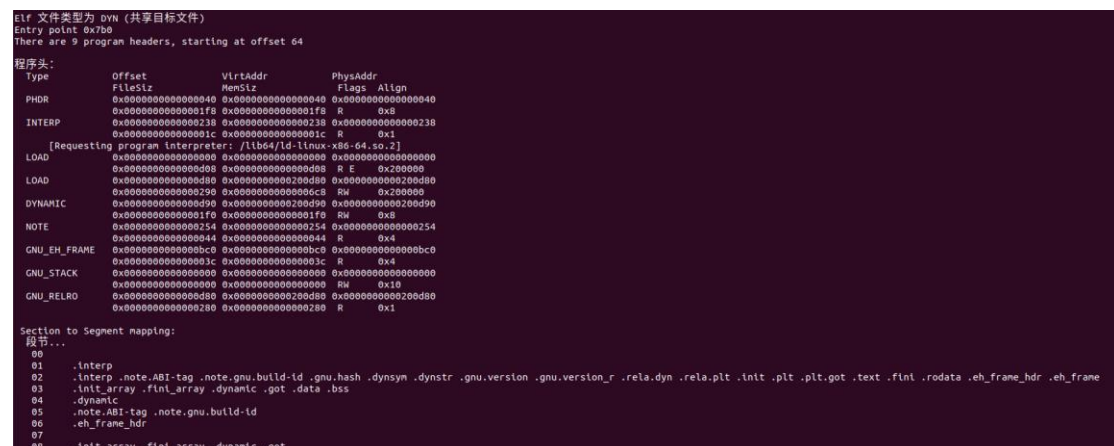
Exercise3: 请自行查阅读取程序头表的指令，然后自行找一个 ELF 可执行文件，读出程序头表并进行适当解释（简单说明即可）。

答：查询 man readelf，不难发现



注意到查询 ELF 文件程序头表的选项是-l

下面进行实验，在 lab1 中，我用 C 语言写了一个可以替代 genBoot.pl 的 genBoot-c1.c 文件，并将其编译为了可执行文件，下面对它进行程序头表分析，执行结果如下：



Type 指对应程序头表表项中的 p_type 项，其中类型有可装入段（LOAD），动态段（DYNAMIC）或解释程序节（INTERP）等。Offset 指出本段首字节在文件中的偏移地址。Vaddr 与 Paddr 分别代表本段在内存中的虚拟地址与物理地址，由于物理地址由操作系统动态分配，因此主要考虑虚拟地址。FileSiz 与 MemSiz

表示本段在 ELF 文件的大小与在内存空间中的大小。Flags 指存储权限。Align 指明对齐方式，用一个模数表示，代表对齐都是 2 的正整数幂。

从下面的 Section to Segment mapping 中可以看到这些段的内容是跟对应 section 的内容重叠的，虽然一个 segment 可能对应多个 section，但是可以根据内存的读写属性、内存特征以及对应段的一般顺序进行区分。

Exercise4: 上面的三个步骤都可以在框架代码里面找得到，请阅读框架代码，说说每步分别在哪个文件的什么部分（即这些函数在哪里）？

答：我们首先按 OS 的启动顺序来确认一下：

1. （步骤一）先是加载 OS 部分（lab1，当然，这次要解析 ELF 格式）。
 - i. ~~从实模式进入保护模式 (lab1)~~
 - ii. ~~加载内核到某地址并跳转运行 (lab1)~~
2. （步骤二）然后是进行系统的各种初始化工作，这里我们采用模块化的方法。

- i. 初始化串口输出（initSerial）

`#!/kernel/kernel/serial.c`

- ii. 初始化中断向量表（initIdt）

`#!/kernel/kernel/idt.c`

- iii. 初始化 8259a 中断控制器（initIntr）

`#!/kernel/kernel/i8259.c`

iv. 初始化 GDT 表、配置 TSS 段 (initSeg)

`#!/kernel/kernel/kvm.c`

v. 初始化 VGA 设备 (initVga)

`#!/kernel/kernel/vga.c`

vi. 配置好键盘映射表 (initKeyTable)

`#!/kernel/kernel/keyboard.c`

vii. 从磁盘加载用户程序到内存相应地址 (loadUMain)

`#!/kernel/kernel/kvm.c`

3. (步骤三) 最后进入用户空间进行输出。

i. 进入用户空间 (enterUserSpace)

`#!/kernel/kernel/kvm.c`

ii. 调用库函数, 输出各种内容!

Exercise5: 是不是思路有点乱? 请梳理思路, 请说说“可屏蔽中断”, “不可屏蔽中断”, “软中断”, “外部中断”, “异常”这几个概念之间的区别和关系。(防止混淆)

答: 解释见下:

1、可屏蔽中断: 指通过可屏蔽中断请求线 INTR 向 CPU 进行请求的中断, 主要来自 I/O 设备。CPU 可以通过在中断控制器 PIC 中设置相应的屏蔽字来屏蔽或不屏蔽它。

2、不可屏蔽中断: 通常是非常紧急的硬件故障, 通过专门的不可屏蔽中断请求

线 NMI 向 CPU 发出中断请求，入电源掉电，硬件线路故障等，该类中断信号一旦产生，不可屏蔽，必然送到 CPU

3、软中断：陷阱是预先安排的一种异常，提供用户程序与内核之间的接口：系统调用，记为 INT N，其中 N 为系统调用号。执行该指令引起的异常就是软中断。

4、外部中断：指 CPU 外部的 I/O 程序向 CPU 发送的打断程序运行的请求。

5、异常：指 CPU 内部执行指令时发生打断程序运行的情况。

Exercise6：这里又出现一个概念性的问题，如果你没有弄懂，对上面的文字可能会有疑惑。请问：IRQ 和中断号是一个东西吗？它们分别是什么？（如果现在不懂可以做完实验再来回答。）

答：IRQ（Interrupt ReQuest）与中断号不是同一个东西。

中断号是向量中断方式下，给予每个异常或中断唯一的编号。在实模式下，这些中断号作为向量索引帮助 CPU 访问 00000H~003FFH 内存中的中断向量表中，表中的每个表项存放异常或中断服务处理程序的入口地址；而在保护模式下，CPU 通过中断描述符表（IDT）访问中断或异常服务程序。

而 IRQ 中文翻译应该为“中断请求”，但实际上一些文档中说到 IRQ 时一般就是指“中断请求信号线”——8259A 的一个引脚，一个 8259A 有 8 个这样的引脚 IRQ0~IRQ7。一个 IRQ 信号线将对应一个中断号，当一个 IRQ 接收到一个外部中断信号后，CPU 向 8259A 发送 INTA 信号，要求被告知 IRQ 对应的中断号，然后根据中断号到 IDT 中找到相应的中断向量，跳转到对应的中断处理子程序入口地址开始执行。

对于 IRQ 而言，其对应的中断号是可编程的。默认情况下，IRQ 对应的中断号是由 BIOS 初始化的，IRQ0~IRQ7 对应着中断号 0x8~0xF。而可编程表现在，我们可以将 IRQ0 对应的中断号修改成 0~255 中的任意一个数，我们设为 INT_NUM，然后 IRQ1~IRQ7 对应的中断号就跟着变成 INT_NUM+1~INT_NUM+7 了。举个例子，我们将 IRQ0 对应的中断号改到 0x10，那么 IRQ1 对应的中断号就是 0x11、IRQ2 的是 0x12……IRQ7 的是 0x17。

Exercise 7: 请问上面用斜体标出的“一些特殊情况”是指什么？

答：一般的软件程序运行需要依赖 EIP，CS 与 EFLAG 指示运行指令的内存位置（EIP，CS）以及相关的状态（EFLAG），这就势必会改变这三个寄存器的值，与保存现场的目的相违背，因此必须用硬件保存这三个寄存器的值。

Exercise8: 请简单举个例子，什么情况下会被覆盖？（即稍微解释一下上面那句话：“如果选择把信息保存在一个固定的地方，发生中断嵌套的时候，第一次中断保存的状态信息将会被优先级高的中断处理过程所覆盖”）

答：假设我们将所有中断的现场保护信息都存放在同一个位置，那么中断嵌套将导致信息的覆盖。

举个例子，比如我在程序运行中调用了一次 INT N，软中断进入了内核态，这时候发生的中断事件需要保护现场信息。然而假如在内核态运行的过程中又发生了一次中断或异常事件，比如除零，缺页等，就需要中断此时内核态的运行，

再一次保护现场，去处理新的这个优先级更高的中断事件。两次现场的保护就会导致第一次的信息被第二次的信息覆盖，使得操作系统无法正确处理中断嵌套的情形。因此需要引入栈结构来保护现场，以支持中断嵌套。

Exercise9: 请解释我在伪代码里用“???”注释的那一部分,为什么要 pop ESP 和 SS?

答：对于 iret 过程中的这一步骤：

```
1. old_CS = CS
2. pop EIP
3. pop CS
4. pop EFLAGS
5. if(GDT[old_CS].DPL < GDT[CS].DPL)    //???
6.     pop ESP
7.     pop SS
```

其中 old_CS 代表返回过程中返回前的 CS 值，而经过 pop CS 后 CS 中存放的是返回目标代码的 CS 值。因此 if(GDT[old_CS].DPL < GDT[CS].DPL)意味着硬件返回是从低特权等级返回到高特权等级。

而从低特权等级返回到高特权等级意味着是从高特权等级切换到低特权等级，注意到此时在切换时，有如下过程：

```
1. if(target_CPL < GDT[old_CS].DPL)    //特权级检查!!!（请翻找手册前面）
2.     TSS_base = GDT[TR].base          //检查通过，获得 TSS 段的基地址（结构体地址）
3.     switch(target_CPL)                //我要切换到 ring 几（在本实验中只有 ring0）
4.         case 0:
5.             SS = TSS_base->SS0
6.             ESP = TSS_base->ESP0
7.         case 1:
8.             SS = TSS_base->SS1
```



```

9.          ESP = TSS_base->ESP1
10.      case 2:
11.          SS = TSS_base->SS2
12.          ESP = TSS_base->ESP2
13.      push old_SS
14.      push old_ESP

```

注意标红的两句，此时是在低特权等级的堆栈中 push 了高特权等级的堆栈位置，那么从低特权等级返回高特权等级时，也需要从当前堆栈中取出目标堆栈的位置，即 pop SS, pop ESP。

Exercise10: 我们在使用 eax, ecx, edx, ebx, esi, edi 前将寄存器的值保存到了栈中（注意第五行声明了 6 个局部变量），如果去掉保存和恢复的步骤，从内核返回之后会不会产生不可恢复的错误？

答：会。因为进入内核态之后，如果在软中断前不保存这些寄存器的值，内核态中如果修改了相应寄存器的值，返回用户态之后就无法恢复现场，可能影响到程序的后续运行。

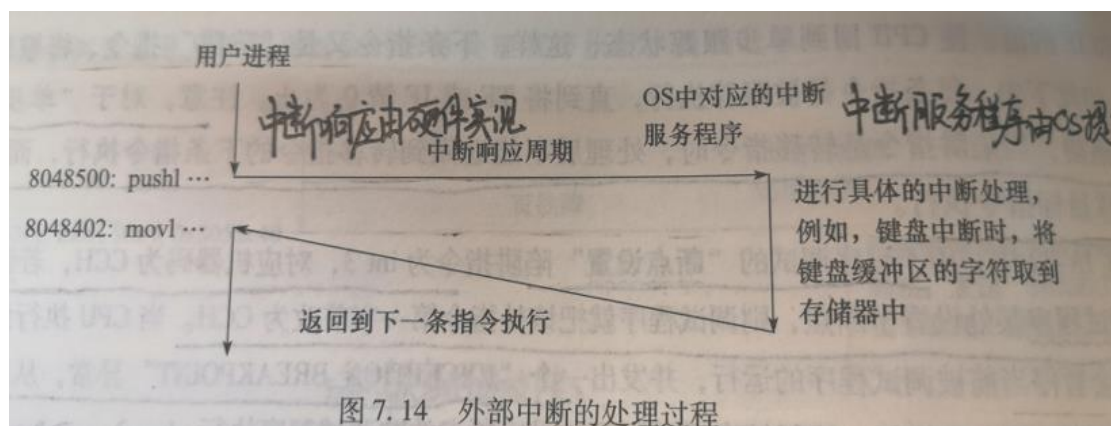
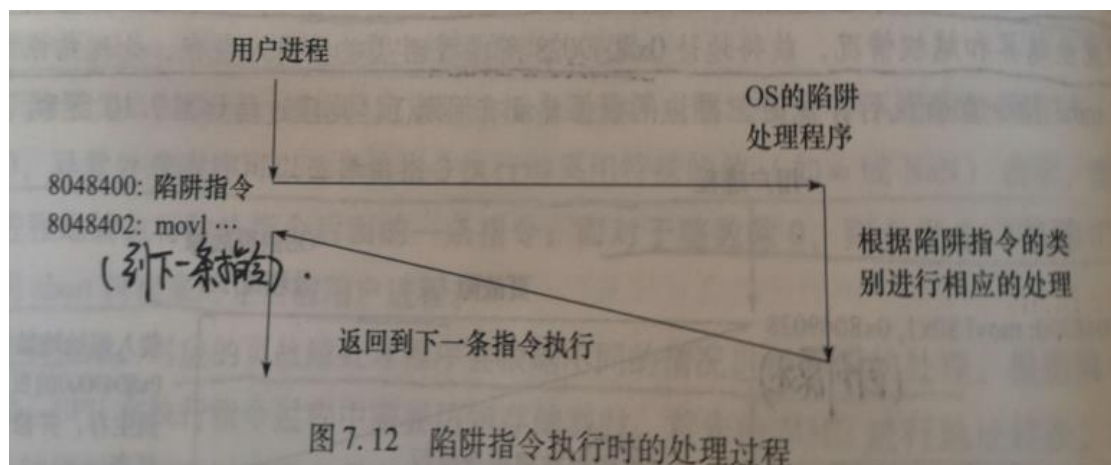
当然我认为这六个寄存器保存的逻辑还是有些区别的，eax, ecx, edx 是 caller 保存的寄存器，这是替 INT N 这个软中断指令保存的；而 ebx, esi, edi 是 callee 保存的寄存器，是替 syscall 的调用者保存的。

其中最重要的寄存器是 %ebp，如果栈帧底的地址信息缺失了，函数的返回值就找不到了，程序的运行显然无法正常进行。

Exercise11: 我们会发现软中断的过程和硬件中断的过程类似，他们最终都会

去查找 IDT，然后找到对应的中断处理程序。为什么会这样设计？（可以做完实验再回答这个问题）

答：因为软中断（Trap）与硬件中断有着相同的处理流程与应对逻辑。那么为了简化设计，完全可以在逻辑上将这种一致的处理流程统一起来，用相同的方法实现。下面两张摘自 ics 教材的图可以说明：



可见软中断与外部中断的处理流程是一致的，都是调用中断服务程序后返回下一条指令继续执行。

Task1: 填写 boot.c，加载内核代码

Exercise 12 为什么要设置 esp？不设置会有什么后果？我是否可以把 esp 设

置成别的呢？请举一个例子，并且解释一下

答：设置 esp 是为了在 bootloader 中初始化栈。跳入 bootMain 运行需要一定的栈空间。不设置的话，esp 可能会是一个随机的值，栈空间中的行为可能会错误地修改代码段与数据段，在运行过程中有会因此出错。

Esp 的值未必一定是 0x1fffff。经过尝试发现，esp 的最小合法值是 0x119020。通过 readelf 分析 kernel 的可执行文件，发现 kernel 段在内存中是从 0x100000 开始的，需要装载的代码段与数据段载 0x104e00 结束，因此栈顶肯定需要比这个地址高出若干个字节以保证程序的正确运行，否则栈中的操作可能会错误地改变数据段乃至代码段的值。

至于为什么初始这设置成 0x1fffff，通过阅读 GDT 的设计，可以看到在 initSeg 开启分段之后用户段的基址是 0x200000，内核态的基址则是 0，因此为了使用户段与内核段互相不冲突，设计内核态的栈顶不超出 0x200000，是比较合理的。在 initSeg 函数中 TSS 中的 esp0 也是设置的 0x1fffff。

当然实际上超出一点也不要紧，考虑到程序的运行过程，此时 gdt，idt，tss 等结构还未初始化，用户程序也为装载，栈只要在进入用户程序前不影响到 kernel 的运行就可以了。

Exercise 13 与 Challenge 1:

粗略编辑以下 kernel，得到 kMain.elf

Readelf -l kMain.elf 考察程序头表，得到如下结果：

1. Elf 文件类型为 EXEC (可执行文件)
2. Entry point 0x1000a8

3. There are 3 program headers, starting at offset 52

4.

5. 程序头:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x001000	0x00100000	0x00100000	0x011a8	0x011a8	R E	0x1000
LOAD	0x003000	0x00103000	0x00103000	0x00120	0x01f00	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10

10.

11. Section to Segment mapping:

12. 段节...

00	.text .rodata .eh_frame
01	.got.plt .data .bss
02	

但是当我用代码分析 elf 的方式完成以后, make 发现代码段达到了 596bytes, 并且我也没有什么把这一部分压缩到 510bytes 以下的好办法。所以我通过 readelf -h kMain.elf 得到了这一 elf 的相关信息, 然后在代码里面手动填写变量值。下面是 ELF 头的信息:

1.	ELF 头:	
2.	Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
3.	类别:	ELF32
4.	数据:	2 补码, 小端序 (little endian)
5.	版本:	1 (current)
6.	OS/ABI:	UNIX - System V
7.	ABI 版本:	0
8.	类型:	EXEC (可执行文件)
9.	系统架构:	Intel 80386
10.	版本:	0x1
11.	入口点地址:	0x1000a8
12.	程序头起点:	52 (bytes into file)
13.	Start of section headers:	34372 (bytes into file)
14.	标志:	0x0
15.	本头的大小:	52 (字节)
16.	程序头大小:	32 (字节)
17.	Number of program headers:	3
18.	节头大小:	40 (字节)
19.	节头数量:	18
20.	字符串表索引节头:	17

逻辑上的代码应当是这样的:

```
1. unsigned char buffer[200*512];
```

```

2.  elf = (void*)buffer;
3.  for (i = 0; i < 200; i++)
4.  {
5.      readSect((void*)(elf + i * 512), 1 + i);
6.  }
7.  ELFHeader* eh=(ELFHeader*)elf;
8.  kMainEntry=(void*)(void))(eh->entry);
9.  phoff=eh->phoff;
10. ProgramHeader* ph=(ProgramHeader*)(elf+phoff);
11. ProgramHeader* phEnd=ph + eh->phnum -1;
12. for(; ph <= phEnd; ph++){
13.     if(ph->type == PT_LOAD){ // PT_LOAD == 1
14.         offset = ph -> off;
15.         unsigned int filesz = ph -> filesz;
16.         unsigned int memsz = ph -> memsz;
17.         unsigned int vaddr = ph -> vaddr;
18.         for( i = 0; i < filesz; i++){*(unsigned char*)(vaddr + i) =
            *(unsigned char*)(elf + offset + i) ;
19.         for( i = 0; i < memsz - filesz; i++) *(unsigned char*)(vaddr +
            filesz + i) = 0;
20.     }
21. }

```

这其中需要注意，之所以创建一个局部变量 unsigned char buffer[200*512]，是因为框架代码中本来设置的是 elf = 0x100000，而这里比较特殊的是 LOAD 段需要装载到的目的地址敲好为 0x100000,这样在装载的时候会覆盖程序头表，使得后续对于 ph 的访问出错。因此为了避免在这个问题上出错，额外开辟空间 buffer 作为程序头表的拷贝，防止出现覆盖问题。

阅读程序头表的内容，一共有两个需要装填的段。而其中数据段已经在需要装填的位置了 (elf + offset = vaddr)，只需要装填代码段即可,并且显然代码段的 filesz == memsz。这样就可以压缩代码大小。改成如下：

```

1. int i = 0;
2.     int phoff = 0x34;
3. unsigned int elf = 0x100000;
4. void (*kMainEntry)(void);
5. kMainEntry = (void (*)(void))0x100000;
6. int offset = 0x1000;

```

```

7.
8. for (i = 0; i < 200; i++)
9. {
10.     readSect((void *)(elf + i * 512), 1 + i);
11. }
12.
13. // TODO: 填写 kMainEntry、phoff、offset..... 然后加载 Kernel (可以参考
    NEMU 的某次 lab)
14. // If we use codes to analyse this elf, the boot block will be too la
    rge.
15. // So just use readelf to analyse elf outside, and fill the variables
    by hand.
16. // Load .text only.
17. ELFHeader *eh = (ELFHeader *)elf;
18. kMainEntry = (void (*)(void))(eh->entry);
19. ProgramHeader *ph = (ProgramHeader *)(elf + phoff);
20. for (i = 0; i < ph->filesz; i++)
21.     *(unsigned char *)(elf + i) = *(unsigned char *)(elf + offset + i
        );

```

至于错误代码为什么可以正确运行，查询 elf 中的程序头表。首先按照错误代码的写法，代码段与只读数据段实际上是被装载正确的了，因此运行指令不会出错。0x103000 往后到 0x104e00 的可读可写数据段虽然会被错误地装载，但由于此时 kMain.Elf 中不存在.data 节，只有未赋初值的.bss 节，因此所有全局变量都不会受错误装载的影响。唯一受到影响的是.got.plt，这两个表跟重定位延迟绑定有关，而本 elf 文件中有没有重定位文件，因此错误代码歪打正着地可以正确地运行。

Task 2: 完成 Kernel 的初始化

Exercise14: 请查看 Kernel 的 Makefile 里面的链接指令，结合代码说明 kMainEntry 函数和 Kernel 的 main.c 里的 kEntry 有什么关系。kMainEntry 函数究竟是个啥？

答: \$(LD) \$(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf \$(KOBJS)

将 kEntry()函数设置为程序的入口函数, 作用等同于默认情况下的 main 函数。因此 kMain.elf 中的 entry 就是 kEntry()函数的地址。所以, 实际上 kMainEntry 与 kEntry 描述了同一个函数入口, 也就是记录了同一个地址。

Task 3: 初始化 IDE

Exercise15:到这里,我们对中断处理程序是没什么概念的,所以请查看 doirq.S, 你就会发现 idt.c 里面的中断处理程序, 请回答: 所有的函数最后都会跳转到哪个函数? 请思考一下, 为什么要这样做呢?

答: 都会跳转到 asmDoIrq 函数中去。这是一个中断服务处理函数, 传入一个作为参数的 TrapFrame* tf 后, asmDoIrq 会根据 tf->irq (即该中断或陷阱事件对应的中断向量号) 调用相应的中断处理程序。这样做为所有的中断处理过程提供了统一的接口, 提高代码的效率与可读性。

Exercise16:请问 doirq.S 里面 asmDoirq 函数里面为什么要 push esp? 这是在做什么?

答: 这个%esp 是作为 asmDoIrq(TrapFrame* tf)的参数传入的。观察到在调用 asmDoIrq 之前, 首先会向栈中压入一个整型数作为 tf->irq, 随后的 pusha 将寄存器的值依次保存到栈中, 这样栈中就等价于保存了一个 TrapFrame 变量, 其首地址是由当前的%esp 指出的。

Exercise17: 请说说如果 keyboard 中断出现嵌套, 会发生什么问题? (Hint: 屏幕会显示出什么? 堆栈会怎么样?)

答: 假如键盘作为 Trap 出现嵌套, 那么输出的循序就会被打乱, 因为 CPU 会依次将被打断的字符输入的寄存器现场压入栈, 并从栈顶向下开始处理。比方讲输入 ab 两个字母, 在 a 的输入还未处理结束时, 就输入了 b, 那么 CPU 将暂停处理 a 的输入, 并将其寄存器现场压入栈中, 转而处理 b 的输入; 等 b 的输入处理结束, 再处理 a 的, 结果得到的输入顺序变成了 ba。

Exercise18: 阅读代码后回答, 用户程序在内存中占多少空间?

答: 根据以下两句可以看到:

```
1. gdt[SEG_UCODE] = SEG(STA_X | STA_R, 0x00200000, 0x000fffff, DPL_USER);  
2. gdt[SEG_UDATA] = SEG(STA_W, 0x00200000, 0x000fffff, DPL_USER);
```

用户空间从 0x00200000 开始, limit 为 0xfffff, 即共占 0x100000 字节的内存空间。

Task 4: 填充中断处理程序, 保证系统在发生中断时能合理处理

Exercise19: 请看 syscallPrint 函数的第一行:

```
int sel = USEL(SEG_UDATA);
```

请说说 sel 变量有什么用。

答: 主要看函数中这三条指令:

```
1. char *str = (char*)tf->edx;  
2. asm volatile("movw %0, %%es"::"m"(sel));
```



```
3. asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
```

第一条指令用一个局部变量 `str` 记录参数 `tf` 保存在 `edx` 中的需要输出的字符串的首地址，但是需要注意。但是需要注意这个首地址值是在用户段的偏移量。在一般的 Linux 环境下，所有段的 `base` 都是 0，但是在框架代码中，用户段的基地址是 `0x00200000`，而内核段的基地址却是 0。当前 `ds` 记录的是内核段的基地址（`syscall` 之后程序在内核态运行），所以通过 `str` 访问首地址需要知道用户态的首地址，第二第三条指令就是完场这一工作。

第二条内联汇编指令将 `sel` 变量的值移动到 `es` 中，而第三条指令就是通过在用户态下记录用户态数据段信息的段选择子 `es` 与字符串首地址 `str`，下标 `i` 一起访问当前循环中需要输出的字符，将其赋值给 `character` 变量（`character` 存储在某一寄存器中）。

因此，`sel` 是在框架代码内核态、用户态基地址不同的情况下，在内核态访问用户态信息的媒介。

Challenge2：比较关键的几个函数

KeyboardHandle 函数是处理键盘中断的函数

syscallPrint 函数是对应于“写”系统调用

syscallGetChar 和 syscallGetStr 对应于“read”系统调用

有以下两个问题

1、请解释框架代码在干什么。

答：解释参照代码注释

(1) `KeyboardHandle` 负责从键盘接收字符的输入

```

1. extern int displayRow; //光标所在的行
2. extern int displayCol; //光标所在的列
3.
4. extern uint32_t keyBuffer[MAX_KEYBUFFER_SIZE]; //ascii 码队列
5. extern int bufferHead; //队列头
6. extern int bufferTail; //队列尾
7.
8. int tail=0; //定义行首位置
9. void KeyboardHandle(struct TrapFrame *tf){
10.     uint32_t code = getKeyCode(); //得到键盘码, 函数定义在 keyboard.c
        中
11.
12.     if(code == 0xe){ // 退格符
13.         // 要求只能退格用户键盘输入的字符串, 且最多退到当行行首
14.         if(displayCol>0&&displayCol>tail){
15.             displayCol--;
16.             uint16_t data = 0 | (0x0c << 8);
17.             int pos = (80*displayRow+displayCol)*2;
18.             asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000))
                ;
19.
20.         }
21.     }else if(code == 0x1c){ // 回车符
22.         // 处理回车情况
23.         keyBuffer[bufferTail++]='\n';
24.         displayRow++;
25.         displayCol=0;
26.         tail=0;
27.         if(displayRow==25){ //处理滚屏
28.             scrollScreen();
29.             displayRow=24;
30.             displayCol=0;
31.         }
32.     }else if(code < 0x81){ // 正常字符
33.         // 注意输入的大小写的实现、不可打印字符的处理
34.         char character=getChar(code); //实现键盘码到 ascii 码的转换
35.         if(character!=0){
36.             putChar(character); //put char into serial
37.             uint16_t data=character|(0x0c<<8); //构造向 vga 输出的 16 位
                字符信息, 后八位是 ascii 码, 前 8 位是颜色等信息
38.             //将 ascii 字符存入字符队列以备取用
39.             keyBuffer[bufferTail++]=character;
40.             bufferTail%=MAX_KEYBUFFER_SIZE;
41.             //在 VGA 中输出该字符

```

```

42.         int pos=(80*displayRow+displayCol)*2;
43.         asm volatile("movw %0,(%1)":"r"(data),"r"(pos+0xb8000));
44.         displayCol+=1; //光标列数加1
45.         if(displayCol==80){ //考虑换行（若有必要，考虑滚屏）
46.             displayCol=0;
47.             displayRow++;
48.             if(displayRow==25){
49.                 scrollScreen();
50.                 displayRow=24;
51.                 displayCol=0;
52.             }
53.         }
54.     }
55. }
56.     //更新光标信息
57.     updateCursor(displayRow, displayCol);
58.
59. }

```

(2) syscallPrint 函数是对应于“写”系统调用

```

1. void syscallPrint(struct TrapFrame *tf) {
2.     //syscallPrint 负责字符串的输出
3.     int sel = USEL(SEG_UDATA); //segment selector for user data, need
        further modification //在内核段记录用户段的段选择符
4.     char *str = (char*)tf->edx; //从 tf->edx 中取出待输出字符串在用户段中
        的地址（偏移量）
5.     int size = tf->ebx; //从 tf->ebx 中取出待输出字符串的大小
6.     int i = 0;
7.     int pos = 0;
8.     char character = 0;
9.     uint16_t data = 0;
10.    asm volatile("movw %0, %%es":"m"(sel));
11.    for (i = 0; i < size; i++) {
12.        //用内联汇编将用户段的 str[i]赋值到局部变量 character
13.        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i))
            ;
14.        // 完成光标的维护和打印到显存
15.
16.        if(character!='\n'){ //输出普通字符
17.            data = character | (0x0c << 8);
18.            pos = (80*displayRow+displayCol)*2;
19.            asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000))
                ;
20.            displayCol+=1;

```

```

21.         if(displayCol==80){
22.             displayCol=0;
23.             displayRow++;
24.             if(displayRow==25){
25.                 scrollScreen();
26.                 displayRow=24;
27.                 displayCol=0;
28.             }
29.         }
30.
31.     }
32.     else{ //输出换行符
33.         displayCol=0;
34.         displayRow++;
35.         if(displayRow==25){
36.             scrollScreen();
37.             displayRow=24;
38.             displayCol=0;
39.         }
40.     }
41. }
42. tail=displayCol; //记录此时的行尾
43. updateCursor(displayRow, displayCol); //更新光标
44. }

```

(3) syscallGetChar 和 syscallGetStr 对应于“read”系统调用

```

1. extern int displayRow; //光标所在的行
2. extern int displayCol; //光标所在的列
3. extern uint32_t keyBuffer[MAX_KEYBUFFER_SIZE]; //ascii 码队列
4. extern int bufferHead; //队列头
5. extern int bufferTail; //队列尾
6.
7. void syscallGetChar(struct TrapFrame *tf){
8.
9.     bufferHead=0;
10.    bufferTail=0;
11.    keyBuffer[bufferHead]=0;
12.    keyBuffer[bufferHead+1]=0; //初始化字符队列
13.    char c=0;
14.
15.    while(c == 0){
16.        enableInterrupt(); //开中断(sti)，等待键盘外设中断调用
        keyboardHandle 接收字符
    }
}

```

```

17.         c = keyBuffer[bufferHead]; //若有字符，从字符队列 keyBuffer 头读
           取
18.         putChar(c); //将输入的字符输出到串口
19.         disableInterrupt(); //关中断(cli)，目的在于防止输入字符的速度过
           快，使得中断嵌套
20.     }
21.     tf->eax=c; //返回值
22.
23.     char wait=0; //等待输入结束
24.     while(wait==0){
25.         enableInterrupt();
26.         wait = keyBuffer[bufferHead+1];
27.         disableInterrupt();
28.     }
29.     return;
30. }
31.
32. void syscallGetStr(struct TrapFrame *tf){
33.     char* str=(char*)(tf->edx); //str pointer
34.     int size=(int)(tf->ebx); //str size
35.     bufferHead=0;
36.     bufferTail=0;
37.     for(int j=0; j<MAX_KEYBUFFER_SIZE; j++) keyBuffer[j]=0; //init
38.     int i=0;
39.
40.     char tpc=0;
41.     while(tpc!='\n' && i<size){
42.
43.         while(keyBuffer[i]==0){ //开中断，等待键盘输入
44.             enableInterrupt();
45.         }
46.         tpc=keyBuffer[i]; //将输入的结果存入字符队列
47.         i++;
48.         disableInterrupt();
49.     }
50.
51.     int selector=USEL(SEG_UDATA);
52.     asm volatile("movw %0, %%es"::"m"(selector));
53.     int k=0;
54.     for(int p=bufferHead; p<i-1; p++){ //将字符队列中的数据转移至目标字
           串
55.         asm volatile("movb %0, %%es:(%1)"::"r"(keyBuffer[p]), "r"(str+
           k));
56.         k++;

```

```

57.     }
58.     asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
59.     return;
60. }

```

2、阅读前面人写的烂代码是我们专业同学必备的技能。很多同学会觉得这三个函数给的框架代码写的非常烂，你是否有更好的实现方法？可以替换掉它

答：其实我觉的还行，毕竟上学期我自己写的数电实验的代码就未必好到哪去。

但是我觉得可以把 VGA 输出的代码封装起来，封装成三个专门负责向 VGA 外设输出的内联函数，像下面这样：

```

1. // vga functions only handle displayRow and displayCol
2. inline void vgaNewLine(){
3.     displayCol=0;
4.     displayRow++;
5.     if(displayRow==25){
6.         scrollScreen();
7.         displayRow=24;
8.         displayCol=0;
9.     }
10. }
11.
12. inline void vgaPutChar(char c){
13.     uint16_t data = c | (0x0c << 8);
14.     uint32_t pos = (80*displayRow+displayCol)*2;
15.     asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
16.     displayCol+=1;
17. }
18.
19. inline void vgaBackSpace(){
20.     displayCol--;
21.     uint16_t data = 0 | (0x0c << 8);
22.     int pos = (80*displayRow+displayCol)*2;
23.     asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
24. }

```

在下面的函数中，需要 VGA 输出的部分直接调用这三个函数就可以了，就不用相同的代码在几个函数里复制了几次，感觉逻辑上会更加清晰一点。

还有一个问题，keyboard 里面还套了 vga 跟 serial 输出的逻辑，耦合度感觉

太高了，两者分开来可能更清晰一些，但是这样要改的太多了，就不做修改了，毕竟能跑就行，也并不是特别难读。

Task 5： 完善库函数

Exercise20: paraList 是 printf 的参数，为什么初始值设置为\&format?

假设我调用 printf("%d = %d + %d", 3, 2, 1);, 那么数字 2 的地址应该是多少?

所以当我解析 format 遇到%的时候，需要怎么做?

答：根据栈的特征，2 的地址是 paraListt[2]，因此遇到%就 index++

exercise21: 关于系统调用号，我们在 printf 的实现里给出了样例，请找到阅读这一行代码 syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0); 说一说这些参数都是什么（比如 SYS_WRITE 在什么时候会用到，又是怎么传递到需要的地方呢？）。

答：SYS_WRITE 保存在%eax 中，STD_OUT 保存在%ecx 中，buffer 保存在%edx 中，count 保存在%ebx 中，它们将在调用 irqHandle 前被压入栈，作为函数的参数 TrapFrame* tf 的一部分在相关函数中被调用。

SYS_WRITE 在 syscallHandle 中被调用，将选择 syscallWrite(tf);函数。

SYS_OUT 在 syscallWrite 中被调用，将选择 syscallPrint(tf);函数。

Buffer 与 count 则将在 syscallPrint 函数中被调用。

Exercise21: 关于系统调用号，我们在 printf 的实现里给出了样例，请找到阅

读这一行代码

```
syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
```

说一说这些参数都是什么（比如 SYS_WRITE 在什么时候会用到，又是怎么传递到需要的地方呢？）。

答：SYS_WRITE 是在 kernel/kernel/irqHandle.c/syscallHandle() 中负责选择 syscallWrite；STD_OUT 负责在 syscallWrite 中选择 syscallPrint；buffer 是需要输出的字符串的首地址；count 是字符串的长度。这些参数会在 lbs/syscall.c/syscall() 中在 int 0x80 之前保存在各个寄存器中，然后在 kernel/kernel/dolrq.S/asmDolrq() 中被 pusha 指令压入栈，构成 trap frame 的一部分，传入 syscallHandle 函数中做参数。

Exercise22：记得前面关于串口输出的地方也有 putchar 和_putstr 吗？这里的两个函数和串口那里的两个函数有什么区别？请详细描述它们分别在什么时候能用。

答：串口输出是向外设串口输出信息，在 Makefile 中加上 -serial stdio 后，程序会将 qemu 模拟的串口数据即时输出到 stdio（即宿主机的标准输出）。而 irqHandle.c 中的两个函数则是利用系统调用 INT \$0x80 向 VGA 输出。

串口输出在用户态与内核态都可以调用，本质上就是汇编 out 指令向内存中的串口 SERIAL_PORT 0x3f8 位置进行操作。

而 irqHandle.c 中的这两个函数则是将通过软中断从用户态切换到内核态，在调用向模拟的 VGA 外设输出。在 syscall.c 中，通过对 syscall 进行封装，得到

了可以在用户态调用的 printf, getChar, getStr 等函数。

Challenge3：如果你读懂了系统调用的实现，请仿照 printf，实现一个 scanf 库函数。并在 app 里面编写代码自行测试。最后录一个视频，展示你的 scanf。

（写出来加分，不写不扣分）

答：这里贴一个 scanf 的代码实现，具体视频见附件：

```
1. int str2Int(char *string){
2.     int flag = 1;
3.     int res = 0;
4.     if(*string == '-'){
5.         flag = 0;
6.         string++;
7.     }
8.     while(*string){
9.         res *= 10;
10.        res += *(string++) - '0' ;
11.    }
12.    return flag?res:-res;
13. }
14.
15. void scanf(const char* format,...){
16.     int i=0; // format index
17.     char buffer[MAX_BUFFER_SIZE] = {'\0'};
18.     int index=0; // parameter index
19.     void *paraList=(void*)&format; // address of format in stack
20.     int state=0; // 0:' legal character'; 1: '%'; 2: illegal format
21.     while(format[i] != '\0'){
22.         switch(state){
23.             case 0:{
24.                 if(format[i++] == '%') state = 1;
25.                 else state = 2;
26.                 break;
27.             }
28.             case 1:{
29.                 switch(format[i++){
30.                     case 'd': {
```

```

31.             getStr(buffer,MAX_BUFFER_SIZE);
32.             *((int**)paraList)[++index] = str2Int(buffer)
               ;
33.             break;
34.         }
35.         case 'c':{
36.             *((char**)paraList)[++index] = getChar();
37.             break;
38.         }
39.         case 's':{
40.             getStr(((char**)paraList)[++index],MAX_BUFFER
               _SIZE);
41.             break;
42.         }
43.         default:state = 2;break;
44.     }
45.     state = 0;
46.     break;
47. }
48. default:{
49.     return;
50. }
51. }
52. }
53. }

```

Exercise23：请结合 gdt 初始化函数，说明为什么链接时用"-Ttext 0x00000000"参数，而不是"-Ttext 0x00200000"。

答：app.c 中的用户进程函数在 elf 的地址是用户代码段与用户数据段的偏移量，其首地址就是从 0 开始；而 0x00200000 是用户段的基地址，记录在 GDT 中，与用户进程 ELF 中的地址无关。在内核态装载 uMain.elf 函数，此时基地址为 0，因此需要将 uMain.elf 的代码段装载到 0x00200000 开始的内存空间里去，再进入用户态运行程序。

Conclusion 1: 请回答以下问题

请结合代码，详细描述用户程序 app 调用 printf 时，从 lib/syscall.c 中 syscall 函数开始执行到 lib/syscall.c 中 syscall 返回的全过程，硬件和软件分别做了什么？（实际上就是中断执行和返回的全过程，syscallPrint 的执行过程不用描述）

答：1、软件保存 6 个寄存器的值，将需要的参数填入对应的寄存器中，通过 INT \$0x80 进入内核态。

2、执行 INT \$0x80，硬件保存 CS, EIP, EFLAGS 的值，并通过检测到的中断服务号 0x80，从 IDTR 指向的 IDT 中取出第 0x80 个表项 IDT0x80(在 initIdt()中初始化)。

3、硬件根据 IDT0x80 中的段选择符，从 GDTR 中取出相应的段描述符，得到对应异常中断程序的所在段的 DPL、基址等信息。具体在代码框架中，指 setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);中定义的信息，即程序将转入 irqSyscall 执行。

4、此时需要硬件检查 CPL 与 DPL，将从用户态转入内核态。因此，应从用户栈切换到内核栈，通过以下步骤完成栈的切换：

(1) 读 TR 寄存器，以访问正在运行的 TSS 段

(2) 将 TSS 段中保存的内核栈的段选择符和栈指针分别装入寄存器 SS 与 ESP，然后在内核栈中保存原来的用户栈 SS 与 ESP。

2、3、4 这三个过程全部由硬件完成。

4、执行 irqSyscall（定义在 doIrq.S 中），压入出错码与中断服务号之后跳转到 asmDoIrq，期间通过 pushal 将寄存器情况保存到栈中，构造 TrapFrame*，随后

跳转到 irqHandle() 执行，并跳转至 syscallHandle(TrapFrame*)。这一步属于可编
程中断服务，由软件执行。

5、软件执行 syscallWrite，并转入 syscallPrint 执行。

6、函数执行完毕，逐个返回（硬件负责处理 CS，EIP，SS，ESP，EFLAGS 与特权
等级的转换，其余都由软件负责）。

具体而言，硬件需要在终端过程中完成的部分入手册中如下部分所述：

```
1. ##### int i #####
2. old_CS = CS
3. old_EIP = EIP
4. old_SS = SS
5. old_ESP = ESP
6. target_CS = IDT[vec].selector //target_cs 是中断向量为 vec 的中断
   描述符里的 selector
7. target_CPL = GDT[target_CS].DPL //target_CPL 是目标代码段的 DPL
8. if(target_CPL < GDT[old_CS].DPL) //特权级检查!!!
9.     TSS_base = GDT[TR].base //检查通过，获得 TSS 段的基地址（结构
   体地址）
10. switch(target_CPL) //我要切换到 ring 几（在本实验中只有
   ring0）
11.     case 0:
12.         SS = TSS_base->SS0
13.         ESP = TSS_base->ESP0
14.     case 1:
15.         SS = TSS_base->SS1
16.         ESP = TSS_base->ESP1
17.     case 2:
18.         SS = TSS_base->SS2
19.         ESP = TSS_base->ESP2
20.     push old_SS
21.     push old_ESP
22.     push EFLAGS //把原来的 eflags, cs, eip 都 push 进
   堆栈（目标特权等级的堆栈，比如用户态向内核态的转换，就是 ring0 的堆栈）
23.     push old_CS
24.     push old_EIP
25.
26. ##### iret #####
27. //这个和前面的流程顺序上.....
28. old_CS = CS
```

```

29. pop EIP
30. pop CS
31. pop EFLAGS
32. if(GDT[old_CS].DPL < GDT[CS].DPL)    ///???
33.     pop ESP
34.     pop SS

```

Conclusion2: 请回答下面问题: 请结合代码, 详细描述当产生保护异常错误时, 硬件和软件进行配合处理的全过程。

答: 当出现保护异常之后, 硬件检查检查发现 $CPL < DPL$, 抛出 #GP, 硬件将终止程序执行, 转而从 IDT 中取出表项 IDT13, 跳转入 irqGProtectFault 函数执行 (定义在 dolrq.S 中)。随后压入错误码与中断服务号号之后转入 asmDolrq 函数执行, 一次将寄存器的值压入栈, 创造 TrapFrame* tf, 并跳入 irqHandle(TrapFrame*) 执行 (定义在 irqHandle.c 中)。随后根据 tf->irq 中的中断服务号, 选择进入 GProtectFaultHandle(tf), 然后 abort() 终止程序。

Conclusion 4 一点意见与一个问题:

1、串口的 putNum 写错了, 按原本的写法输出的数字实际上是反的, 比如 56 将输出成 65。修改后具体见下方代码:

```

1. void putNum(int num){
2.     static char buf[30] = { 0 };
3.     char* p = buf + sizeof(buf) - 1;
4.     if (num == 0){ putChar('0'); return;}
5.     if (num < 0){ putChar('-'); num = -num;}
6.     while(num){
7.         *--p = (num % 10) + '0';
8.         num /= 10;
9.     }
10.    while(*p) putChar(*(p++));

```

```
11. }
```

2、修改了 abort.c, 里面的功能完全可以通过调用 putchar 与_putstr 实现, 重复的相同逻辑代码反而使得程序不好读:

```
1. #define BLUE_SCREEN_TEXT "Assertion failed: "  
2. static void displayMessage(const char *file, int line) {  
3.    _putstr((char*)BLUE_SCREEN_TEXT);  
4.    _putstr((char*)file);  
5.     putchar(':');  
6.     putNum(line);  
7.     putchar('\n');  
8. }
```

3、在实验的过程中, 发现 irqHandle 函数中 tf->irq==-1 时需要直接 break 而不是 assert(0)。这意味着在运行的过程中有一些中断函数没有实现。但是从 idt 中取出表项的过程是硬件完成的, 因此我没有什么合适的方法去调试, 只能根据二分法试出来出错的两个中断服务号是 46 跟 48, 也就是说只要一下中断服务号 46 跟 48 被调用了, 但是没有定义它们的处理函数。可是查表可知, 这两个应该是自定义的软中断, 不知道在什么地方被调用了。