

# Lab2 实验报告

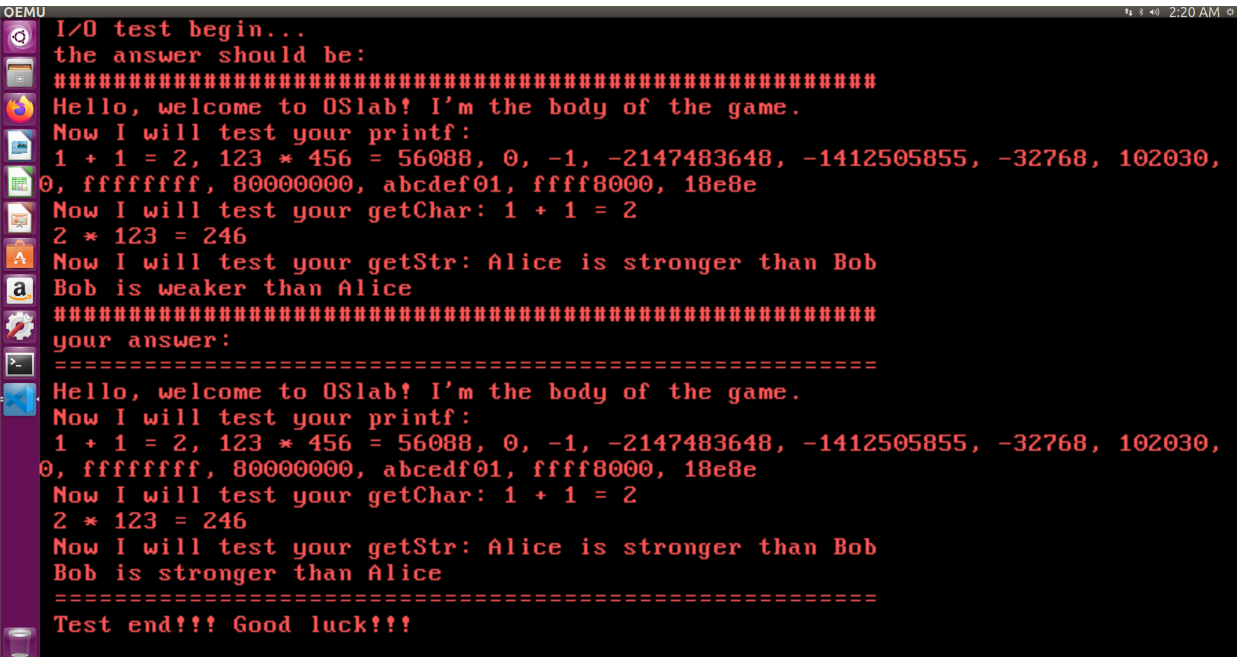
课程名称：操作系统      任课教师：叶保留

|       |  |         |                    |
|-------|--|---------|--------------------|
| 学院    | 计算机科学与技术系  | 专业      | 计算机科学与技术           |
| 学号    | 221220016  | 姓名      | 殷子珺                |
| Email | <a href="mailto:221220016@smail.nju.edu.cn">221220016@smail.nju.edu.cn</a> | 开始/完成日期 | 2024/4/7-2024/4/18 |

## 实验进度

我完成了所有内容

## 实验结果



```
OEMU
I/O test begin...
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030,
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030,
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is stronger than Alice
=====
Test end!!! Good luck!!!
```

## 实验修改代码位置

- `bootloader/boot.c` : `bootMain()` 函数, 加载内核代码
- `kernel/main.c` : `kEntry()` 函数, 完成Kernel的初始化
- `kernel/idt.c` : `setIntr()` , `setTarp()` , `initIdt()` 函数, 完成IDT相关初始化
- `kernel/irqhandle.c` : `irqHandle()` , `syscallPrint()` , `syscallGetChar()` , `syscallGetStr()` 函数, 完善中断处理
- `lib/syscall.c` : `printf()` , `getchar()` , `getStr()` 函数, 完善输入输出库函数
- `kernel/kvm.c` : `loadUMain()` 函数, 加载用户程序
- `Makefile` : 修改了部分编译选项 (不会对未使用的变量报错)

(部分代码实现呈现在以下的实验思考中, 不再单独叙述每一部分代码的具体实现)

# 实验思考

(参考[操作系统lab2【耳听八方-中断机制】](#) | [OSlab\(gitbook.io\)](#)中部分思考题)

## 1. EFLAGS、CS、EIP 只能由硬件保存的原因

- **原子性**：当中断发生时，处理器需要立即停止当前的指令流并跳转到中断处理程序。如果这三个寄存器的值由软件保存，那么在保存这些寄存器的值的过程中可能会发生另一个中断，这可能导致这些寄存器的值被错误地保存
- **EIP寄存器**：当中断发生时，处理器需要保存EIP寄存器的值，以便在中断处理完成后能够返回到被中断的位置。由于EIP寄存器的值在每条指令执行后都会改变，所以这个寄存器的值必须在中断发生的那一刻由硬件来保存
- **CS和EFLAGS寄存器**：CS寄存器保存了当前代码段的选择子，EFLAGS寄存器保存了处理器的状态，如果这两个寄存器的值在中断处理过程中被改变，那么在中断处理完成后，程序可能无法正确地执行

## 2. start.S 中 esp 设置为0x1fffff的原因

编译时内核 `.text` 段的起始地址设为 `0x100000`，内核加载至物理内存 `0x100000` 开始的位置；编译时用户程序 `.text` 段的起始地址设为 `0x200000`，用户程序加载至物理内存 `0x200000` 开始的位置，则 `0x1fffff` 恰好为内核栈的栈顶，设置完 `esp` 后执行加载 `kernel` 的程序，使用到的栈区为内核栈。

实际上如果只是为了正确执行加载 `kernel` 的程序，可以不需要指定 `esp` 指向内核栈栈顶的位置，只要能保证 `esp` 指向的地址是可写的，且足够大以容纳堆栈所需的空间，就能作为合理的 `esp`。在后续 `kernel` 初始化 `TSS` 时也会将 `esp0` 设置在内核栈顶的位置

```
//相关代码
start.S
    movl $0x1fffff,%eax      # setting esp
    movl %eax,%esp

kernel/kernel/kvm.c initSeg()
    //初始化TSS
    tss.esp0 = 0x1fffff;
    tss.ss0 = KSEL(SEG_KDATA);
    asm volatile("ltr %%ax:: \"a\" (KSEL(SEG_TSS));");
```

## 3. 关于GitHub上 bootMain 函数实现的错误

(我最开始没有头绪也是照着GitHub上的代码写的，在此纠正错误)

```

//错误代码
// TODO: 填写kMainEntry、phoff、offset
ELFHeader* eh=(ELFHeader*)elf;
kMainEntry=(void(*) (void))(eh->entry);
phoff=eh->phoff;
ProgramHeader* ph=(ProgramHeader*)(elf+phoff);
offset=ph->off;

for (i = 0; i < 200 * 512; i++) {
    *(unsigned char *) (elf + i) = *(unsigned char *) (elf + i + offset);
}
kMainEntry();

```

想要理解出现的错误，首先需要回顾 **程序头表** 的相关内容：

在可执行文件中，ELF头、程序头表、`.init` 节、`.fini` 节、`.text` 节和 `.rodata` 节合起来可构成一个只读代码段，`.data` 节和 `.bss` 节合起来可构成一个可读写数据段。显然，在可执行文件启动运行时，这两个段必须装入内存且需要为之分配存储空间，因而称为可装入段 (`PT_LOAD`)。

但是程序头表除了可装入段以外，还有其他类型的段，以下进行验证：

用 `readelf -l a.out` 命令打印出可执行文件 `hello` 的程序头表，可以看到程序头表的第一项就不是需要被装载的段，也就说在 `ELFHeader` 中读到的 `Start of program headers: 64 (bytes into file)`，即 `phoff` 项并不是直接指向一个可装入段。

```
C hello.c x
C hello.c > ...
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello, World!");
5     return 0;
}

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

• yimo@ubuntu:~/code$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz             Flags   Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000001f8 0x00000000000001f8  R E     8
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
               0x000000000000001c 0x000000000000001c  R      1
 [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000006fc 0x00000000000006fc  R E    200000
LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x0000000000000228 0x0000000000000230  RW    200000
DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
               0x00000000000001d0 0x00000000000001d0  RW     8
NOTE          0x0000000000000254 0x0000000000400254 0x0000000000400254
               0x0000000000000044 0x0000000000000044  R      4
GNU_EH_FRAME   0x00000000000005d4 0x00000000004005d4 0x00000000004005d4
               0x0000000000000034 0x0000000000000034  R      4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000  RW    10
GNU_RELRO     0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x00000000000001f0 0x00000000000001f0  R      1

No Ports Forwarded
```

此时若直接使用 `elf + phoff->Offset` 作为磁盘中可装入段程序开始被装入的位置，就会从程序头表中的第一个表项，即类型为 `PHDR` 的表项开始，将内容移动到 `entry` 指示的程序入口处，这样 `entry` 处并不是可以执行的代码段，甚至不是需要被装入内存的段。

那为什么这个错误的程序确可以正确地运行呢？这肯定要怀疑是程序头表中的表项排布问题，以下进行验证：

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

sed-variable -c -o ../lib/syscall.o ../lib/syscall.c
ld -m elf_i386 -e uEntry -Ttext 0x00000000 -o uMain.elf ./main.o ../lib/syscall.o
make[1]: Leaving directory '/home/yimo/os2024/lab2/app'
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
● yimo@ubuntu:~/os2024/lab2$ readelf -l kernel/kMain.elf

Elf file type is EXEC (Executable file)
Entry point 0x100000
There are 3 program headers, starting at offset 52

Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x001000 0x00100000 0x00100000 0x014ec 0x014ec R E 0x1000
LOAD           0x003000 0x00102000 0x00102000 0x00100 0x01ea4 RW 0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

Section to Segment mapping:
Segment Sections...
00      .text .rodata .eh_frame
01      .data .bss
02
```

的确，在 `kMain.elf` 文件的程序头表中第一项为类型 `LOAD` 的 `.text` 可装入段，直接使用 `elf + phoff -> offset` 阴差阳错地得到了正确的可装入段位置，从而让 `kMainEntry` 函数正确跳转至 `kernel` 代码处执行。并且由上图的信息可以看出，代码框架中原本初始化 `kMainEntry`、`phoff`、`offset` 时已经给出了正确的初始位置，即使不再对这三个变量进行任何赋值都能正确执行

```
//写在框架代码中的部分
int phoff = 0x34; // program header offset
int offset = 0x1000; // .text section offset
unsigned int elf = 0x100000; // physical memory addr to load
void (*kMainEntry)(void);
```

以下为修改后的代码，注释掉的 `for` 循环实现部分编译后会使得引导块 `boot` 的大小超过了最大限制510字节（最后两个字节被保留用于存储引导签名 `0xAA55`），所以改用 `while` 循环更简洁地实现

```
#define PT_LOAD 1
// TODO: 阅读boot.h查看elf相关信息，填写kMainEntry、phoff、offset
ELFHeader *elfHeader = (struct ELFHeader *)elf;
kMainEntry = (void (*)(void))elfHeader->entry;
phoff = elfHeader->phoff;
ProgramHeader* ph = elf + phoff;
ProgramHeader* eph = ph + elfHeader->phnum;
while(ph->type != PT_LOAD)
    ph++;
offset = ph->off;

/*for( ; ph < eph; ph++) {
    if(ph->type == PT_LOAD) {
        offset = ph->off;
        ph = eph;
    }
}*/
```

#### 4. kMainEntry 和 kEntry 之间的关系

根据 kernel/Makefile 文件中的链接指令 `-e kEntry -Ttext 0x00100000 -o kMain.elf $(KOBJS)` 得出 kEntry 为程序入口点，也就是内存地址 0x100000 处；而 kMainEntry 是设置为指向 0x100000 内存地址的函数指针，通过在 bootMain 中调用 kMainEntry()，可以直接跳转至相应地址执行 kEntry() 的第一条可执行指令。

#### 5. asmDoIrq 函数中的 push ebp

```
asmDoIrq:
    pushal // push process state into kernel stack
    pushl %esp //esp is treated as a parameter
    call irqHandle

void irqHandle(struct TrapFrame *tf) //irqHandle function signature

struct TrapFrame {
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    int32_t irq;
};
```

当中断来临时，中断处理的全过程分为硬件和软件（操作系统）处理两部分：

- 硬件处理
  - 根据中断向量号找到门描述符，再根据是否门描述符中的 selector 找到中断程序对应的段描述符（均为内核代码段），确定是否需要堆栈切换，若要切换则将原来的SS和ESP保存在新栈
  - 依次将EFLAGS、CS、EIP压栈
  - 若有Error Code也压栈
  - 依据门描述符装载 CS 与 EIP，即执行中断处理程序
- 操作系统处理（根据 TrapFrame 数据结构进行现场保护）
  - 将中断向量号压栈
  - 按顺序保存所有寄存器
  - 处理阶段/恢复阶段

此时 esp 指向栈顶，即 TrapFrame 中的第一个元素 edi。在调用 irqHandle 函数前，将 esp 作为 TrapFrame \*tf 参数压栈，即可以通过该指针访问整个 TrapFrame 中储存的内容

#### 6. keyboard中断出现嵌套的后果

- 屏幕显示：如果键盘中断处理程序在执行过程中被另一个键盘中断打断，并且这两个中断都试图在屏幕上显示字符，那么可能会出现字符显示混乱的情况。例如，第一个中断可能只显示了一部分字符，然后第二个中断开始显示，导致第一个中断的剩余字符被插入到第二个中断的字符之间

- 堆栈溢出：每次中断发生时，CPU都会将当前的EFLAGS、CS、EIP以及其他一些寄存器的值压入堆栈，然后跳转到中断处理程序。如果中断频繁发生并且嵌套深度过大，可能会导致堆栈溢出。这可能会覆盖和破坏其他内存区域的数据，导致程序错误

所以把 `irqkeyboard` 设置为中断类型，在执行中断处理程序之前会关中断

## 7. 关于 `selector` 填写

//pa中的实现：

```
for (i = 0; i < NR_IRQ; i++)
    set_trap(idt + i, SEG_KERNEL_CODE << 3, (uint32_t)irq_empty, DPL_KERNEL);
set_trap(idt + 0, SEG_KERNEL_CODE << 3, (uint32_t)vec0, DPL_KERNEL);
.....
```

//直接对段的 `index` 部分左移3位组成 `selector`

回顾段选择子（16位）的结构：

```

15          3   1 0
+-----+-----+
|                   | T |  |
|           INDEX   | | RPL |
|                   | I |  |
+-----+-----+
TI - TABLE INDICATOR, 0 = GDT, 1 = LDT
RPL - REQUESTOR'S PRIVILEGE LEVEL
```

可见段选择子低3位并非规定位全0，而是标志位。全局描述符中 `TI = 0`，中断处理程序都存在内核代码段，`RPL = 00`，所以在这个地方可以直接左移3位得到正确的段选择子

在本次实验的代码框架中定义了两个根据段生成段选择子的宏 `KSEL` 和 `USEL`，可以直接调用完成意义上正确且可读性强的实现

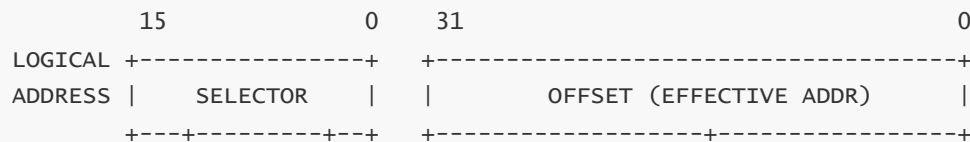
```
/* 为了防止系统异常终止，所有irq都有处理函数(irqEmpty)。 */
for (i = 0; i < NR_IRQ; i++)
    setTrap(idt + i, KSEL(SEG_KCODE), (uint32_t)irqEmpty, DPL_KERN);
/*init your idt here 初始化 IDT 表，为中断设置中断处理函数*/
setTrap(idt + 0x8, KSEL(SEG_KCODE), (uint32_t)irqDoubleFault, DPL_KERN);
```

## 8. 在 `syscallPrint` 和 `syscallGetStr` 函数中切换段选择子的必要性

```
int sel = USEL(SEG_UDATA);
asm volatile("movw %0, %%es" :: "m"(sel));
```

这两个函数都涉及从寄存器中读出一个字符串地址，并读出这个地址中的内容或向这个地址写入一些内容。

在保护模式下，QEMU中运行的程序在访问内存时给出的是由一个16位的段选择符加上32位的段内偏移量（有效地址）所构成的48位的逻辑地址：



所以要在段寄存器中设置正确的段选择子以进行内存访问。此时中断处理程序运行在内核态，对应段寄存器中（应该）是内核段的段选择子（但是没在代码框架中具体看到这一部分的设置），直接根据传进来的 `offset` 地址访问用户态数据时会访问到错误的地方。

而之所以更改的是 `es` 寄存器中的段选择子而不是直接改 `ds` 寄存器的段选择子，是因为 `ds` 寄存器通常用于存储默认的数据段选择子，`es` 寄存器可以用于存储额外的数据段选择子。内核需要访问用户空间的数据，同时也可能需要访问内核空间的数据。如果内核改变了 `ds` 的值，那么它就需要在访问完用户空间的数据后，再将 `ds` 寄存器的值恢复为内核数据的段选择子，增加代码的复杂性以及出错可能性。具体使用哪个段寄存器由编译器决定。

不过在扁平模式下，所有的段基址都为0，实际上换不换也无所谓 ……

## 9. `printf` 中解析格式化字符串的两种实现

方法一：使用 `va_list` 相关宏实现

```
typedef char *va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, format) (ap = (va_list) & format + _INTSIZEOF(format))
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
#define va_end(ap) (ap = (va_list)0)
```

- `_INTSIZEOF(n)`：用于计算类型 `n` 的大小，并将其向上取整到最接近的 `int` 大小的倍数。这是因为在大多数系统中，参数是以 `int` 或更大的单位（如 `double` 或指针）来传递的，即使参数的实际类型可能更小（如 `char` 或 `short`）
- `va_start(ap, format)`：将 `ap` 指向可变参数列表的第一个参数
- `va_arg(ap, type)`：用于从参数列表中获取下一个参数。它首先使用 `_INTSIZEOF(type)` 来计算下一个参数在内存中的位置，然后将 `ap` 指针向前移动相应的距离，然后返回该位置的值。这样做的目的是确保 `ap` 指针总是指向下一个参数的开始位置，即使参数的大小是 `int` 的整数倍



```

paraList += sizeof(char *); //指向format之后的第一个参数
.....
switch (format[i])
{
    case 'd':
        decimal = va_arg(paraList, int); //每次调用va_arg即可获取参数列表中的
下一个参数
        count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
        state = 0;
        break;
    .....
}

```

## 方法二：直接对paraList进行调整

```

paraList += sizeof(char *); //指向format之后的第一个参数
.....
switch (format[i])
{
    case 's':
        string = *(char **)paraList; //将paraList转换成相应类型的指针
        paraList += 4; //参数的大小为int的整数倍
        count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
        state = 0;
        break;
    .....
}

```

补充：paraList 本身是 (void \*) 类型的参数，指向 const char\* format，所以指向的每个元素的大小为1个字节，执行 paraList++ 会将指针向后移动一个字节，所以在上述代码实现中需要在或许当前参数后将 paraList += 4

## 11. 用户程序的链接地址

```

umain.bin: $(UOBJS)
    @#$(LD) $(LDFLAGS) -e uEntry -Ttext 0x00200000 -o uMain.elf $(UOBJS)
    $(LD) $(LDFLAGS) -e uEntry -Ttext 0x00000000 -o uMain.elf $(UOBJS)

```

在 app/Makefile 中，链接时用 "-Ttext 0x00000000" 参数，而不是 "-Ttext 0x00200000"（提前规定了编译时用户程序 .text 段的起始地址设为 0x200000，用户程序加载至物理内存 0x200000 开始的位置）

可以看到，在 kernel/kvm.c 中

```
#define SEG(type, base, lim, dpl) (SegDesc)

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

在加载 `gdt` 的过程中，将用户代码段和数据段的基地址直接设置为 `0x200000`，所以链接时只需要设置链接选项为 `"-Ttext 0x00000000"` 即可正确装载用户程序

## 实验心得

- 上学期写 `pa` 的时候只是单纯的按要求填空，基本没有好好看过代码框架，对中断机制其实还是一知半解……借这次 `lab` 比较详细地看了所有代码框架，再结合参考了 `pa` 的文档、本实验文档以及计算机系统基础和操作系统基础的课本，更深入地了解了整个中断处理机制的实现，注意到了很多之前没细想过的问题
- 由于 `qemu` 的执行过程并没有直接对我们可见，有些硬件处理的部分的执行结果比较不可预料（比如键盘中断处理中，对全局变量的修改和取值，有时会得到意想不到的结果），这部分实在没法解决，只能当作“系统特性”暂时不予理会了
- `GitHub` 上的过去的 `oslab` 代码正确性有待商榷