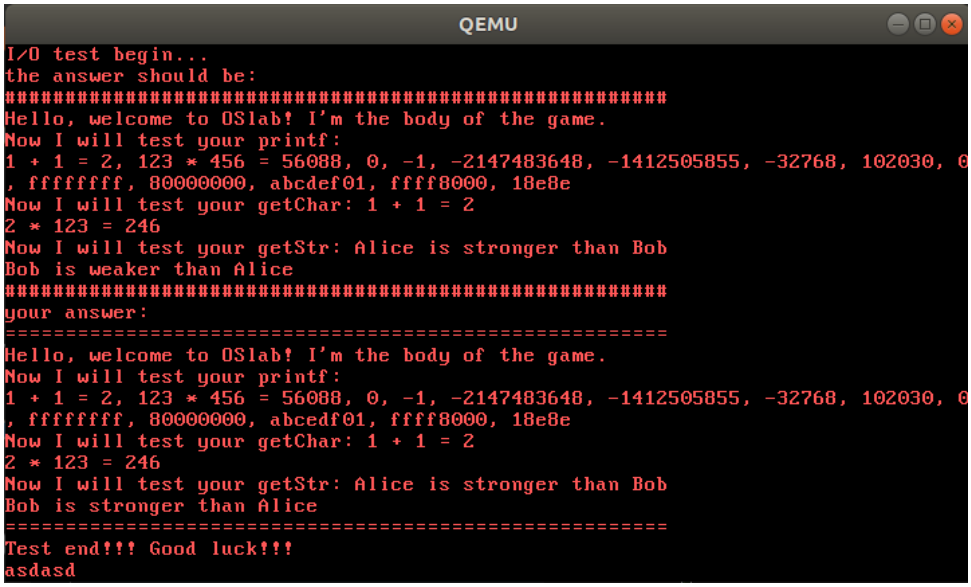
操作系统lab2实验报告

191220156　计算机科学与技术系　张桓　2659428453@qq.com

实验进度

完成了磁盘的加载(`bootloader`加载`kernel`，`kernel`加载用户程序)，实现中断机制，并且完成了`getChar()` `getStr()` `printf()`三个基于中断的函数的**全部内容。**

实验结果



修改内容

跳转到bootloader

setting esp

- 在 `start.S` 中，我们进行了关中断，启动 `A20` 总线，加载 `GDTR` ，启动保护模式，然后设置了多个段选择符，我们参照 `LAB1` 中的框架代码，设置 `kernel` 的栈顶指针 `ESP` 。

```
movl $0x1fffff, %eax            # TODO: setting esp
movl %eax, %esp
```

加载kernel至内存

- 在 `boot.h` 中找到关于 `ELF` 头结构体的定义 `ELFheader` ，框架代码中先用 `readSect()` 函数读磁盘得到 `elf` 文件，加载到内存时，不需要 `elf` 头等不重要的部分，将 `.text` 节 `.data` 节提前即可。下面的 `kMainEntry` 为代码起始位置， `phoff` 为程序头表偏移， `offset` 为 `.text` 偏移。

```
// TODO: 填写kMainEntry、phoff、offset
kMainEntry = (void(*)(void))((struct ELFHeader *)elf)->entry;
phoff = ((struct ELFHeader *)elf)->phoff;
offset = ((struct ProgramHeader *)(elf + phoff))->off;
for (i = 0; i < 200 * 512; i++) {
    *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
}
```

初始化串口

做一系列初始化

```
void kEntry(void) {
    // Interruption is disabled in bootloader
    initSerial();// initialize serial port
    // TODO: 做一系列初始化
    initIdt();          // initialize idt
    initIntr();         // iniialize 8259a
    initSeg();          // initialize gdt, tss
    initVga(); // initialize vga device
    initKeyTable();         // initialize keyboard device
    loadUMain(); // load user program, enter user space
    while(1);
    assert(0);
}
```

初始化IDT

初始化interrupt gate并填好剩下的表项

- 完成设置门函数：在`memory.h`中找到了关于门描述符`GateDescriptor`的定义，按照陷阱门和中断门的结构，利用给好的参数直接填写即可。

```
static void setIntr(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {
    // TODO: 初始化interrupt gate
    ptr->offset_15_0 = offset & 0xFFFF;
    ptr->segment = selector << 3;
    ptr->pad0 = 0;
    ptr->type = INTERRUPT_GATE_32;
    ptr->system = 0;
    ptr->privilege_level = dpl;
    ptr->present = 1;
    ptr->offset_31_16 = (offset >> 16) & 0xFFFF;
}
static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {
    // TODO: 初始化trap gate
    ptr->offset_15_0 = offset & 0xFFFF;
    ptr->segment = selector << 3;
    ptr->pad0 = 0;
    ptr->type = TRAP_GATE_32;
```

```
18        ptr->system = 0;
19        ptr->privilege_level = dpl;
20        ptr->present = 1;
21        ptr->offset_31_16 = (offset >> 16) & 0xFFFF;
22    }
```

- 设置 `IDT` 门描述符，按照 `idt.c` 中声明的函数利用填好的 `setTrap()` 和 `setIntr()` 函数为每个中断服务函数设置好对应的门描述符。

> 要想实现键盘中断处理我们首先要在IDT表中加上键盘中断对应的门描述符，8259a将硬件中断映射到键盘中断的向量号 `0x20 - 0x2F` ，键盘为IRQ1，所以键盘中断号为 `0x21` ，框架代码也提供了键盘中断的处理函数 `irqKeyboard` 。并且由于硬件中断不受DPL影响，8259A的15个中断都为**内核级。**

```
1     // TODO: 填好剩下的表项
2     setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, DPL_KERN);
3     setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent, DPL_KERN);
4     setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault, DPL_KERN);
5     setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
6     setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
7     setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck, DPL_KERN);
8     setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException, DPL_KERN);
9     /* Exceptions with DPL = 3 */
10    // TODO: 填好剩下的表项
11    setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER)
12    setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
```

将irqKeyboard的中断向量号压入栈

```
1   .global irqKeyboard
2   irqKeyboard:
3       pushl $0
4       push $0X21      # TODO: 将irqKeyboard的中断向量号压入栈
5       jmp asmDoIrq
```

实现用户需要的库函数

填好中断处理程序的调用

- 在`memory.h`中找到了关于`TrapFrame`的定义：

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

填好中断处理程序的调用如下：

```
switch(tf->irq) {
        // TODO: 填好中断处理程序的调用
        case -1:
            break;
        case 0xd:
            GProtectFaultHandle(tf);
            break;
        case 0x21:
            KeyboardHandle(tf);
            break;
        case 0x80:
            syscallHandle(tf);
            break;
        default:assert(0);
}
```

完成KeyboardHandle()函数

- `KeyboardHandle()`函数处理一个字符，包含两个功能，首先将该字符保存在`KeyBuffer`缓冲区中，并且将这个字符打印到屏幕上，打印字符我选择在**显存上**实现。
  - 对于键入的不同字符实现不同的操作，我选择在**显存上**实现，将字符直接打印到屏幕上。
    - 退格符：

    由于退格最多只能退到这一行的开头，因此只需要简单判断一下是否`displayCol == 0`，若已经在开头处则不动，不在开头则`displayCol--`

    - 回车符：
      - 输入回车符，需要另起一行，即：`displayRow++; displayCol=0;`。
      - 需要注意的是，当**一页结束**时即`displayRow == 25`，为了维持页面的延续性，下一行再次在`displayRow = 24`处开始。为了使页面呈现滚动效果，再调用封装好的`scrollScreen()`函数。

    > QEMU模拟的屏幕的大小是`80*25`

  - 正常字符：

    首先调用`getChar(code)`通过扫描码得到对应字符，根据教程中的**提示代码**，将字符`character`显示在屏幕的`displayRow`行`displayCol`列。同时注意输出完一个字符后需要**更新光标位置**，这时面临**换行**或**满页**的问题，在上文回车符部分做了详解。

```
void KeyboardHandle(struct TrapFrame *tf){
    uint32_t code = getKeyCode();
    char character;
    uint16_t data;
    int pos = 0;
    if(code == 0xe){ // 退格符
        // TODO: 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
        keyBuffer[bufferTail] = 0 |(0x0c << 8);
        if(bufferTail > 0) bufferTail--;
        if(displayCol > 0 )
            displayCol--;
    }else if(code == 0x1c){ // 回车符
        // TODO: 处理回车情况
        keyBuffer[bufferTail] = '\n';
        bufferTail++;
        displayCol = 0;
        displayRow++;
        if(displayRow == 25)
        {
            displayRow = 24;
            scrollScreen();
        }
    }else if(code < 0x81){ // 正常字符
        // TODO: 注意输入的大小写的实现、不可打印字符的处理
        character = getChar(code);
        if(character >= 32 && character <= 126)
            keyBuffer[bufferTail] = character;
            bufferTail++;
            data = character | (0x0c << 8);
            pos = (80*displayRow+displayCol)*2;
            asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
            displayCol++;
            if(displayCol == 80)//the end of one line
            {
                displayCol = 0;
                displayRow++;
                if(displayRow == 25)//the end of one page
                {
                    displayRow = 24;
                    displayCol = 0;
                    scrollScreen();
                }
            }
        }
    }
    updateCursor(displayRow, displayCol);
}
```

> 查阅资料得知，可见字符的 `ASCII` 码范围为 `32 ~ 126`，因此只有 `character` 在这一范围内的字符才会被显示。

实现printf()函数

用户态的printf()函数

- `printf()` 函数需要对不同的格式 `format` 进行不同的输出处理。通过 `ap` 一个指向字符串的指针，再按照不同的格式不断把移动 `ap` 指针并取4字节内容读取出来，若不足4字节(例如short 和char),那么认为这个元素的大小为4字节，简单的说就是检测元素的大小，不足4字节的当作4字节看待。
  - `%`：如果是 `%` 的话，直接 `count++` 读下一个字符
  - `c`：如果是 `c` 的话，则读出一个字符到 `character` 然后存到 `buffer` 里
  - `s`：如果是 `s` 的话表示一个字符串，则读出字符串的**首地址** `string`，再调用 `str2Str` 将它存在 `buffer` 里
  - `x`：如果是 `x` 的话表示十六进制数，读出来存到 `hexadecimal` 里，再调用 `hex2Str` 存到 `buffer` 里
  - `d`：如果是 `d` 的话表示十进制数，和 `x` 大同小异
- `buffer` 中有内容时系统调用 `syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0)` 输出。有一个情况是如果 `buffer` 已经满了，即 `count==MAX_BUFFER_SIZE`，那么强制调用 `syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0)` 输出。

```
int printf(const char *format,...){
    char* ap;
    ap = (char *)&format+ _INTSIZEOF(format);
    int i=0; // format index
    char buffer[MAX_BUFFER_SIZE];
    int count=0; // buffer index
    int decimal=0;
    uint32_t hexadecimal=0;
    char *string=0;
    char character=0;
    while(format[i]!=0){
        buffer[count] = format[i];
        count++;
        if(format[i]=='%') {
            count--; i++;
            switch(format[i]) {
                case 'c':
                    character=*(char*)((ap += _INTSIZEOF(char)) - _INTSIZEOF(char)) ;
                    buffer[count++]=character;
                    break;
                case 's':
                    string=*(char**)((ap += _INTSIZEOF(char*)) - _INTSIZEOF(char*)) ;
                    count=str2Str(string, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case 'x':
```

```
                    hexadecimal=*(uint32_t*)((ap += _INTSIZEOF(uint32_t)) - _INTSIZEOF(uint32_t));
                    count=hex2Str(hexadecimal, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case 'd':
                    decimal=*(int*)((ap += _INTSIZEOF(int)) - _INTSIZEOF(int));
                    count=dec2Str(decimal, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case '%':
                    count++;
                    break;
            }
        }
        if(count==MAX_BUFFER_SIZE) {
            syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
            count=0;
        }
        i++;
    // TODO: in lab2
    }
    if(count!=0)
        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
    ap = (char *)0;
    return 0;
}
```

> 实现printf()函数时，参考了几篇文章：手写printf函数　prinf函数深度剖析

内核态的syscallPrint()函数

- 首先需要选择用户数据段描述符，`int sel = USEL(SEG_UDATA);`即可

> 刚开始做到这里时我并不是很理解为什么要**切换成用户数据段**，后来在实现`syscallGetstr()`函数时逐渐理解。这是因为当前函数为系统服务例程，是**内核级**的。而调度它的函数为用户级的，传递的地址参数也是**用户空间内**的。因此对于同一个虚拟地址而言，如果直接在内核空间对其寻址的话，会按照内核数据的段描述符来解析出一个线性地址来，而实际的`str`地址是需要按照用户数据段描述符来解析的。因此需要显式切换成用户数阶段，否则将找不到`str`的位置。

- 需要完成光标的维护以及将字符打印到显存
  - 光标维护：
    - 光标的位置`displayRow`和`displayCol`维护只需要**输出一个字符后**`displayCol++`即可
    - 当**一行结束**时即`displayCol == 80`或者**输出换行符**时即`character == '\n'`，需要另起一行，即：`displayRow++; displayCol=0;`

    - 当**一页结束**时即 `displayRow == 25`，为了维持页面的延续性，下一行再次在 `displayRow = 24` 处开始
    - 为了使页面呈现滚动效果，当页面满时需要用封装好的 `scrollScreen()` 函数
- 打印字符：根据教程中的**提示代码**，可以将字符 `character` 显示在屏幕的 `displayRow` 行 `displayCol` 列。

```
void syscallPrint(struct TrapFrame *tf) {
    int sel =  USEL(SEG_UDATA);//TODO: segment selector for user data, need further modification
    ...
    asm volatile("movw %0, %%es"::"m"(sel));
    for (i = 0; i < size; i++) {
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
        // TODO: 完成光标的维护和打印到显存
        if(character != '\n')
        {
            data = character | (0x0c << 8);
            pos = (80*displayRow+displayCol)*2;
            asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
            displayCol++;
            if(displayCol == 80)//the end of one line
            {
                displayCol = 0;
                displayRow++;
                if(displayRow == 25)//the end of one page
                {
                    displayRow == 24;
                    displayCol == 0;
                    scrollScreen();
                }
            }
        }
        else//if the character is \n
        {
            displayCol = 0;
            displayRow++;
            if(displayRow == 25)
            {
                displayRow = 24;
                scrollScreen();
            }
        }
    }
    updateCursor(displayRow, displayCol);
}
```

实现getChar()函数

用户态的getChar()函数

- 这个函数作为 `getChar()` 函数的系统调用，需要将接收到的字符写到 `EAX` 寄存器中**作为返回值**。配合用户态的 `getChar()` 函数一起实现，并且作为 `getChar()` 函数陷入中断后的系统服务例程。
- 可以知道， `getChar()` 函数是作为用户层面的 `API` ，调用该函数后将进行一个系统调度最后执行 `syscallGetchar()` 函数。因此修改 `getChar()` 如下：

```
char getChar(){ // 对应SYS_READ STD_IN
    // TODO: 实现getChar函数，方式不限
    char ch = syscall(SYS_READ, STD_IN, 0, 0, 0, 0);
    return ch;
}
```

内核态的syscallGetchar()函数

- 通过 `syscall()` 函数最终将执行 `syscallGetchar()` 函数，我们知道C标准库里的 `getchar()` 函数是按下字符和回车后，返回字符，如果按了多个字符，那么返回第一个。显然我们需要**关注键盘的事件**，这就用到了 `keyboardHandle()` 函数。因此我们需要**开中断**，从而保证键盘事件的正确监听，这样才能将按下的按键保存在 `keyBuffer` 中。不断观察 `keyBuffer` 中的最新的按键即 `keyBuffer[bufferTail-1]` ，当其为 `\n` 时说明输入结束，这时将输入的第一个字符即 `keyBuffer[bufferHead]` 返回即可。在系统服务例程中返回值放在 `tf->eax` 中，因此修改 `syscallGetchar()` 如下：

```
void syscallGetChar(struct TrapFrame *tf){
    // TODO: 自由实现
    while(1)
    {
        asm volatile("sti");
        if(bufferTail > 0 && keyBuffer[bufferTail - 1] == '\n')
        {
            tf->eax = keyBuffer[bufferHead];
            break;
        }
    }
    bufferTail = 0;
}
```

> 注意到函数最后我还加了句 `bufferTail = 0` ，这是为了清空键盘缓冲区，为了避免下一次调用键盘输入时产生不必要的麻烦，在这里显式进行了缓冲区清空。

实现getStr()函数

用户态的getStr()函数

- 和 `getChar()` 函数类似，该函数也需要配合 `getStr()` 实现。在用户态的 `getStr()` 函数需要系统调用 `syscallGetstr()` 函数，修改如下：

```
void getStr(char *str, int size){ // 对应SYS_READ STD_STR
    // TODO: 实现getStr函数，方式不限
    syscall(SYS_READ, STD_STR, (uint32_t)str, (uint32_t)size, 0, 0);
    return;
}
```

内核态的syscallGetstr()函数

- 而 `syscallGetstr()` 函数得到字符和 `sysycallGetchar()` 函数类似，同样需要**监听键盘事件**，因此**需要开中断**。但是由于 `getChar()` 函数只用关心一个字符，完全可以用 `EAX` 来实现传递。而 `getStr()` 接收的字符串不得不放到一个地址空间即给定的 `char *str` 处。这是在**用户空间内的地址参数**，为了在内核态中能正确访问到这个位置，我们必须**切换到用户数阶段**，这点和 `syscallPrint()` 类似。因此我们也必须用**内联汇编**的方法来对以该地址为首的一系列空间赋值，修改如下：

```
void syscallGetStr(struct TrapFrame *tf){
    // TODO: 自由实现
    int sel =   USEL(SEG_UDATA);//the selector of USER DATA
    asm volatile("movw %0, %%es"::"m"(sel));
    int size = tf->ebx, i = 0;
    char character = 0;
    char *str = (char *)tf->edx;
    while(1)
    {
        asm volatile("sti");
        if(bufferTail > 0)  character = keyBuffer[bufferTail - 1];
        if(character == '\n')   break;
    }
    size = size < bufferTail - 1? size : bufferTail - 1;
    for(i=0; i<size; i++)
    {
        character = keyBuffer[i];
        if(character != '\n')
            asm volatile("movb %0, %%es:(%1)"::"r"(character), "r"(str + i));
        else
            asm volatile("movb %0, %%es:(%1)"::"r"('\0'), "r"(str + i));
    }
    bufferTail = 0;
}
```

> 由于实现方式自由，我这里实现的功能大致对标 `C++` 中的 `gets_s()` 函数。输入回车后代表结束，当输入字符串长度超过 `size` 还没有回车时，直接截断 `0 ~ size-1` 的部分并返回。

加载用户程序至内存

参照bootloader加载内核的方式

- 同加载 `kernel` ，先利用 `readSect()` 函数从磁盘读取 `elf` 文件，再查询 `elf` 文件的**程序头表**，从而得知文件中哪些段需要加载到内存中，将其加载即可。

```
void loadUMain(void) {
    // TODO: 参照bootloader加载内核的方式
    int i = 0;
    int phoff = 0x34;
    int offset = 0x1000;
    uint32_t elf = 0x200000;
    uint32_t uMainEntry = 0x200000;

    for (i = 0; i < 200; i++) {
        readSect((void*)(elf + i*512), 201+i);
    }

    uMainEntry = ((struct ELFHeader *)elf)->entry; // entry address of the program
    phoff = ((struct ELFHeader *)elf)->phoff;
    offset = ((struct ProgramHeader *)(elf + phoff))->off;

    for (i = 0; i < 200 * 512; i++) {
        *(uint8_t *)(elf + i) = *(uint8_t *)(elf + i + offset);
    }
    enterUserSpace(uMainEntry);
}
```

实验心得

✅ 这次实验对我来说还是比较难的，查阅了很多资料，进行了很多尝试。总的来说收获还是很大，特别是对于设计中断调用的函数的内部细节以及中断的实现流程有了更深的理解。比如这次的 `printf()` 函数和 `getStr()` 函数的实现，它们都是在用户态使用的函数，但是不得不通过中断来陷入系统通过调度函数实现功能。

从用户态陷入内核态，**数据段**会发生变化，因此原来在用户态准备的参数，特别是**地址参数**，我们在系统服务例程中没办法直接去使用的。因为在地址解析时会按照内核数据段来解释这个虚拟地址，得到的物理地址将和实际我们需要的物理地址大相径庭。因此才有了**切换段描述符**的做法，将内核数据段切换成用户数据段，这样才能正确得到正确的物理地址。这个做法非常巧妙，令我印象深刻。

- [x] 再有就是对于**软中断和硬中断**的加深理解，这次实验可以说是实现了4个中断过程的完整处理。包括一个硬中断即键盘中断处理和三个软中断即 `printf()` 等三个函数。它们的实现方式和逻辑是完全不一样的，从某种意义上讲这两种中断是平等的。在中断表中分别对应中断号 `0X21` 和 `0X80` 。

  但是键盘中断是**被动的**，按下键盘就会陷入中断执行 `keyboardHandle()` 函数，将按下的字符记录到缓冲区中。而软中断是**主动的**， `syscall()` 函数中显式地写着 `int $0X80` 的语句，再通过提前放在 `EAX` 中的中断号进一步寻找应该做的中断处理程序。

  正是由于软中断和硬中断的“平等”关系，我们在实现诸如 `syscallGetchar()` 的处理程序时，由于需要用到键盘接收的数据，在程序中才会显式地**开中断** `sti` 。开中断后键盘事件能够正常进行，在软中断内部我们再去使用键盘的缓冲区，整个流程就通过中断机制完美实现了。

建议

感觉实验教程的 `html` 格式有点难翻阅，可以换成 `PDF` 之类带目录的格式。再有就是感觉教程里面介绍的内容有点多杂，第一次看的时候一头雾水，可以适当删去一些和实验关联不大的内容，可以放在附录里参考而不是在中途占用大量篇幅，敬供参考🙏

最后感谢助教gg/助教jj的耐心批改😊😊

如有疑问请联系我：2659428453@qq.com