lab2-1718405l4张福翔

联系方式：499480213@qq.com

1. 实验环境

- 自制OS的CPU：Intel 80386
- 模拟80386环境的虚拟机：QEMU
- 交叉编译的编译器：GCC
- 调试工具：GDB
- 运行平台：Ubuntu 18.04
- 编程语言：C，X86 Assembly

2. 实验要求

本实验通过实现一个简单的应用程序，并在其中调用两个自定义实现的系统调用，介绍基于中断实现系统调用的全过程

2.1 实现系统调用库函数`printf`和`scanf`

实验流程如下

1. Bootloader从实模式进入保护模式,加载内核至内存，并跳转执行
2. 内核初始化IDT（Interrupt Descriptor Table，中断描述符表），初始化GDT，初始化 TSS（Task State Segment，任务状态段）
3. 内核加载用户程序至内存，对内核堆栈进行设置，通过`iret`切换至用户空间，执行用户程序
4. 用户程序调用自定义实现的库函数`scanf`完成格式化输入和`printf`完成格式化输出
5. `scanf`基于中断陷入内核，内核扫描按键状态获取输入完成格式化输入(**现阶段不需要考虑键盘中断**)
6. `printf`基于中断陷入内核，由内核完成在视频映射的显存地址中写入内容，完成字符串的打印

2.2 完善`scanf`和`printf`的格式化输入输出

格式化输入输出的测试用例在`lab/app/main.c`中已给出

3. 实验原理

3.1 IA-32中断机制

3.1.1 IDT

中断到来之后，基于中断向量，IA-32硬件利用IDT与GDT这两张表寻找到对应的中断处理程序，并从当前程序跳转执行。

IDT中每个表项称为门描述符（Gate Descriptor），门描述符可以分为3种

- Interrupt Gate，跳转执行该中断对应的处理程序时，`EFLAGS`中的`IF`位会被硬件置为`1`
- Trap Gate，跳转执行该中断对应的处理程序时，`EFLAGS`中的`IF`位不会置为`1`
- Task Gate，Intel设计用于任务切换，现代操作系统中一般不使用

若中断源为`int`等指令产生的软中断，IA-32硬件处理该中断时还会比较产生该中断的程序的CPL与该中断对应的门描述符的DPL字段，若CPL数值上大于DPL，则会产生General Protect Fault，即`#GP`异常。

3.1.2 TSS

中断会改变程序正常执行的流程，为确保中断处理程序执行结束后能正确返回产生中断的程序，IA-32硬件会对产生中断的程序的`EFLAGS`，`CS`，`EIP`等寄存器在堆栈上进行保存

IA-32借助`TR`和TSS来确定保存`EFLAGS`，`CS`，`EIP`这些寄存器信息的新堆栈

`TR`（Task state segment Register）是16位的任务状态段寄存器，结构和`CS`这些段寄存器完全一样，它存放了GDT的一个索引，可以使用`ltr`指令进行加载，通过`TR`可以在GDT中找到一个TSS段描述符。

3.2 系统调用

系统调用的入口定义在`lib`下的`syscall.c`，在`syscall`函数里可以使用嵌入式汇编，先将各个参数分别赋值给`EAX`，`EBX`，`ECX`，`EDX`，`EDX`，`EDI`，`ESI`，然后约定将返回值放入`EAX`中（把返回值放入`EAX`的过程是我们需要在内核中实现的），接着使用`int`指令陷入内核

`int`指令接收一个8-Bits的立即数为参数，产生一个以该操作数为中断向量的软中断，其流程分为以下几步：

1. 查找`IDTR`里面的IDT地址，根据这个地址找到IDT，然后根据IDT找到中断向量的门描述符
2. 检查CPL和门描述符的DPL，如果CPL数值上大于DPL，产生#GP异常，否则继续
3. 如果是一个ring3到ring0的陷入操作，则根据`TR`寄存器和GDT，找到TSS在内存中的位置，读取其中的`SS0`和`ESP0`并装载则向堆栈中压入`SS`和`ESP`，注意这个`SS`和`ESP`是之前用户态的数据
4. 压入`EFLAGS`，`CS`，`EIP`
5. 若门描述符为Interrupt Gate，则修改`EFLAGS`的`IF`位为0
6. 对于某些特定的中断向量，压入Error Code
7. 根据IDT表项设置`CS`和`EIP`，也就是跳转到中断处理程序执行

中断处理程序执行结束，需要从ring0返回ring3的用户态的程序时，使用`iret`指令

`iret`指令流程如下：

1. `iret`指令将当前栈顶的数据依次Pop至`EIP`，`CS`，`EFLAGS`寄存器
2. 若Pop出的`CS`寄存器的CPL数值上大于当前的CPL，则继续将当前栈顶的数据依次Pop至`ESP`，`SS`寄存器
3. 恢复CPU的执行

4. 代码框架

```
lab2-171840514张福翔              #自行修改后打包(.zip)提交
├── lab
|   ├── Makefile
|   ├── app                       #用户代码
|   |   ├── Makefile
```

```
|   |   └── main.c              #主函数
|   ├── bootloader              #引导程序
|   |   ├── Makefile
|   |   ├── boot.c
|   |   ├── boot.h
|   |   └── start.S
|   ├── kernel
|   |   ├── Makefile
|   |   ├── include             #头文件
|   |   |   ├── common
|   |   |   |   ├── assert.h
|   |   |   |   ├── const.h
|   |   |   |   └── types.h
|   |   |   ├── common.h
|   |   |   ├── device
|   |   |   |   ├── disk.h
|   |   |   |   ├── keyboard.h
|   |   |   |   ├── serial.h
|   |   |   |   └── vga.h
|   |   |   ├── device.h
|   |   |   ├── x86
|   |   |   |   ├── cpu.h
|   |   |   |   ├── io.h
|   |   |   |   ├── irq.h
|   |   |   |   └── memory.h
|   |   |   └── x86.h
|   |   ├── kernel              #内核代码
|   |   |   ├── disk.c          #磁盘读写API
|   |   |   ├── doIrq.S         #中断处理
|   |   |   ├── i8259.c         #重设主从8259A
|   |   |   ├── idt.c           #初始化中断描述
|   |   |   ├── irqHandle.c     #中断处理函数
|   |   |   ├── keyboard.c      #初始化键码表
|   |   |   ├── kvm.c           #初始化 GDT 和加载用户程序
|   |   |   ├── serial.c        #初始化串口输出
|   |   |   └── vga.c
|   |   ├── lib
|   |   |   └── abort.c
|   |   └── main.c              #主函数
|   ├── lib                     #库函数
|   |   ├── lib.h
|   |   ├── syscall.c           #系统调用入口
|   |   └── types.h
|   └── utils
|       ├── genBoot.pl          #生成引导程序
|       └── genKernel.pl        #生成内核程序
└── report
    └── lab2-171840514张福翔.pdf
```

5. 解决思路

5.1 中断机制的实现

助教提供的代码框架中已经包含了中断机制的大体实现过程，借助了数据结构`TrapFrame`和封装好的中断处理函数来实现相应的处理，其中数据结构`TrapFrame`的定义如下：

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

我们在执行中断处理程序时，先将中断号压入栈中，再通过`pusha`指令将各个通用寄存器的值压栈，这就形成了一个`TrapFrame`结构，最后通过`push %esp`将`TrapFrame`数据结构的指针压入栈中，调用函数`irqHandle`，即把`TrapFrame`结构指针传参给了该函数。随后，通过文件`irqHandle.c`中的`irqHandle`函数，利用 C 语言根据不同的中断指令，进行不同的处理。我们这里主要考虑系统调用中断（0x80）的处理过程，即操作系统提供的系统调用服务例程：

```
    switch(tf->irq) {
        case -1:
            break;
        case 0xd:
            GProtectFaultHandle(tf);
            break;
        case 0x80:
            syscallHandle(tf);
            break;
        default:
            assert(0);
    }
```

另外，根据 Linux 系统调用的格式，我们规定好对应于`printf`和`scanf`所需要使用的读写对应调用号：

```
#define SYS_WRITE 0
#define SYS_READ 1
#define STD_OUT 0
#define STD_IN 1
```

根据调用号，我们即可调用对应的`syscallPrint`和`syscallScan`函数，分别用于将一个`buffer`输出到标准输出流和从标准输入流读入一个字符串到`buffer`中。

5.2 `printf`函数实现

框架代码中已经提供系统调用例程`syscallPrint`的实现代码，即将字符串中的内容依次写入到显存当中。我们只需考虑在`printf`函数中如何调用这样的系统服务例程，并根据格式化字符`%`来替换对应的字符。

我们先来考虑`printf`的函数声明：

```
int printf(const char *format, ...)
```

这是一个具有可变参数长度的函数，其第一个参数是一个字符串`format`，后面跟着一系列的与格式化字符串对应的参数，假设后面的参数有`a`、`b`、`c`，那么它们在栈中的位置如下：

```
|---------|
|    c    |
|---------|
|    b    |
|---------|
|    a    |
|---------|
|  format |
|---------|   <- %ebp + 8 == const char * format
```

因此，我们只需根据`format`中的格式化符号个数，从栈中找出对应的参数即可，即：

```
void *paraList = (void *)(&format) + 4; // address of format in stack
```

指针`paraList`对应`format`的后一个参数指针，即格式化参数的起始位置。

然后，我们只需遍历`format`字符串，对其中的普通字符直接存放到要打印的字符串`buffer`里，对于格式化字符`%`则根据其类型，找到相对应的参数，并调用已提供好的辅助函数`dec2Str`、`hex2Str`、`str2Str`来将其转化为字符串并存入到`buffer`中。

```
    while(format[i]!=0){
        // TODO: support more format %s %d %x and so on
        if (format[i] == '%')
        {
            i++;
            switch (format[i])
            {
                case 'd':
                    decimal = *((int *)(paraList + 4 * index));
                    index++;
                    count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
                    break;
                case 'x':
                    hexadecimal = *((uint32_t *)(paraList + 4 * index));
                    index++;
                    count = hex2Str(hexadecimal, buffer, MAX_BUFFER_SIZE, count);
                    break;
                case 's':
                    string = *((char **)(paraList + 4 * index));
                    index++;
                    count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
                    break;
                case 'c':
                    character = *((char *)(paraList + 4 * index));
                    index++;
                    buffer[count++] = character;
                    break;
                default:
```

```
            // default case: read the next character, it may differ from the std printf
                    buffer[count++] = format[i];
                    break;
                }
            }
            else
            {
                buffer[count++] = format[i];
            }
            if(count==MAX_BUFFER_SIZE) {
                syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
                count=0;
            }
            i++;
        }
        if(count != 0)
            syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
```

在遍历`format`的过程中，如果`buffer`已经满了，我们需要先通过`syscall`将当前的`buffer`内容先输出到标准输出流中，然后再用清空的`buffer`继续操作。`format`遍历完成后，我们检查`buffer`内是否还有数据需要输出，如果有的话则再进行一次`syscall`打印余下的字符。

代码中的变量`i`和`count`分别对应当前字符串`format`和`buffer`的索引指针，而变量`index`则记录了格式化参数的个数。

5.3 `scanf`函数实现

5.3.1 中断服务例程`syscallScan`的实现

中断服务例程`syscallScan`需要从标准输入流读入一段信息，并将其转为字符串返回，这需要我们通过轮询的方法从键盘中读取一个个字符，直到输入回车`'\n'`字符结束，下面这段代码实现了这个过程：

```
    // TODO: get key code by using getKeyCode and save it into keyBuffer
    uint32_t code;
    do
    {
        code = getKeyCode();
        if (getChar(code) == '\n')
        {
            keyBuffer[bufferTail] = code;
            bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
            break;
        }
        if (code != 0)
        {
            keyBuffer[bufferTail] = code;
            bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
        }
    } while (1);
```

键盘外设提供的函数接口`getKeyCode`返回当前的有效键码，如果键码无效则返回`0`，这是我们不需要存入键盘`buffer`中的。由此，所有输入的键码将存入数组`keyBuffer`中，我们通过框架提供的`getChar`函数将其转化为对应的`ASCII`字符值。

```
    while(i < size-1){
        if(bufferHead != bufferTail){
            character = getChar(keyBuffer[bufferHead]);
            bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
            putChar(character);
            if(character != 0){
                asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
                i++;
            }
        }
        else
            break;
    }
```

我们利用一条硬编码汇编指令`movb`将转化的`char`字符存入字符串中，实现所需的要求，另外，`putChar`函数提供了一个与串口外设交互的方式，以便我们检验输入的字符是否正确。

5.3.2 库函数`scanf`的实现

理论上说，函数`scanf`的`format`参数中任何非格式化字符都是几乎没有意义的，类似于`printf`，我们应该在标准输入流中**原样输入**这些非格式化字符，在我们的实现中，如果检查到输入不符合`format`中的非格式化字符，就会直接终止`scanf`函数，并返回当前的格式化参数个数。

```
int scanf(const char *format, ...)
```

同样的，`scanf`也是一个具有变长参数的函数，它的参数在栈中的存储情况与`printf`完全相同。

我们对`format`字符串进行遍历，并对格式化字符调用相应的辅助函数进行操作。需要注意的是，`scanf`的格式化参数都是对应变量的指针，我们需要修改的是该指针的内容。

```
    while (format[i])
    {
        if (flag)
        {
            if (format[i] >= '0' && format[i] <= '9') // 格式化输入宽度
            {
                limit = 0;
                for (int k = i; format[k] >= '0' && format[k] <= '9'; k++)
                {
                    limit = limit * 10 + format[k] - '0';
                }
            }
            else
            {
                switch (format[i])
                {
                    case 'd':
```

```
                    dec = *(int **)(paraList + index * 4);
                    str2Dec(dec, buffer, MAX_BUFFER_SIZE, &j);
                    index++;
                    break;
                case 'x':
                    hex = *(int **)(paraList + index * 4);
                    str2Hex(hex, buffer, MAX_BUFFER_SIZE, &j);
                    index++;
                    break;
                case 'c':
                    cha = *(char **)(paraList + index * 4);
                    *cha = buffer[j];
                    index++;
                    j++;
                    break;
                case 's':
                    str = *(char **)(paraList + index * 4);
                    str2Str2(str, limit, buffer, MAX_BUFFER_SIZE, &j);
                    index++;
                    break;
                default:
                    j++;
                    break;
            }
            flag = 0;
            limit = MAX_INT;
        }
    }
    else
    {
        if (format[i] == '%')
        {
            flag = 1;
            matchWhiteSpace(buffer, MAX_BUFFER_SIZE, &j); // 去除输入前的空格
        }
        else
        {
            if (format[i] != ' ' && format[i] != '\t' && format[i] != '\n')
            {
                matchWhiteSpace(buffer, MAX_BUFFER_SIZE, &j);
                if (format[i] != buffer[j])
                    return index;
                j++;
            }
        }
    }
    i++;
}
return index;
```

在检查到格式化字符时，我们通过函数`matchWhiteSpace`来去除`buffer`中数据之前的空白内容，然后根据格式化字符类型来调用不同的辅助函数。如果没有检查到格式化字符，我们就直接将`buffer`的指针`j`向前移。最后，我们返回格式化参数的个数，`scanf`函数就此完成。值得注意的是，如果我们在发现格式化字符的同时，在却发现`buffer`中已经读取到尽头，这表明我们标准输入流的输入还没有达到所需的参数规模，因此需要再次进行一次系统调用，从标准输入流中获取新的`buffer`。