实验五：系统调用

1 实验题目

规范化中断处理，实现一组基本的系统调用，并制作相应的库文件供正常编程使用。
具体来说，共有四项：

1. 修改实验 4 的内核代码，先编写 `save()` 和 `restart()` 两个汇编过程，分别用于中断处理的现场保护和现场恢复，内核定义一个保护现场的数据结构，以后处理程序的开头都调用 `save()` 保存中断现场，处理完后都用 `restart()` 恢复中断现场；
2. 内核增加 `int 20h`、`int 21h` 软中断的处理程序，其中，`int 20h` 用于用户程序结束返回内核准备接受命令的状态；`int 21h` 用于系统调用，并实现 3-5 个简单系统调用功能（如输入输出）；
3. 进行 C 语言的库设计，实现 `putch()`、`getch()`、`gets()`、`puts()`、`printf()`、`scanf()` 等基本输入输出库过程；
4. 利用自己设计的 C 库，编写一个使用这些库函数的 C 语言用户程序，产生COM程序。增加内核命令执行这个程序。

2 实验目的

1. 学习掌握 PC 系统的软中断指令；
2. 掌握操作系统内核对用户提供服务的系统调用程序设计方法；
3. 掌握 C 语言的库设计方法；
4. 掌握用户程序请求系统服务的方法。

3 实验要求

1. 保留无敌风火轮显示，取消触碰键盘显示 OUCH! ；
2. 扩展实验四的的内核程序，增加输入输出服务的系统调用；
3. C 语言的库设计，实现基本输入输出库过程。

4 实验方案

工具与环境：Windows（x86_64），gcc，NASM，ld 2.25.1，bochs 虚拟机（1MB 内存、1.44MB 软驱）

本实验新知识不多，使用上次实验的中断编写技术即可。重难点仍是中断程序的编写规范，包括保护及恢复现场、系统调用的细节及规范、系统再入问题等。

实验过程部分将以流畅的方式叙述实验细节，更多试错过程写在实验总结部分。

5 实验过程

5.1 现场保护和恢复

现编写一个现场保护和恢复的模板过程 save、restart，功能为进入中断后保存 CPU 的所有上下文，离开中断前恢复 CPU 上下文。

一个中断处理的整体结构为：

```
someInterrupt:
        call save
        ; interrupt process
        jmp restart
```

用 jmp 是为了不在内核栈中留下垃圾，在 restart 过程中得到用户地址直接回去。

首先要设计一个记录上下文的数据结构，存放在内核里：

```
typedef struct cpuState{
        int eax,ebx,ecx,edx,ebp,edi,esi;
        short ip,cs,flag,ds,es,ss,sp;
} cpuState;

cpuState tmpCpuState;
```

由于我的 C 代码用的都是 32 位寄存器，故有些寄存器需要记录 32 位的。这个数据结构在以后的多进程实验的“进程控制块”中也会用到。

接下来设计 save。就是用 mov 命令把寄存器的值都保存到 tmpCpuState 变量中。关键之处在于语句顺序的合理排布，不妨考虑如下几个问题：

1、要访问 tmpCpuState，就要先把 ds 更改为内核数据段基址，那么如何使得原本的 ds 不丢失？

2、要更改 ds，就要用 ax 来给它赋值，那么如何使得原本的 ax 不丢失？

3、ip,cs,flag 等都在用户栈中，那么 ss,sp 要等到什么时候才能恢复到内核栈？

由此可见，语句顺序必须被精心设计，以不破坏任何原有数据。

参考 minix.asm[1]，先将 eax,ds 入栈（此时的栈仍是用户程序的栈），然后利用 ax 来修改 ds 为内核数据段基址，接着将所有寄存器的值保存到 tmpCpuState 中（其中 eax,ds,ip,cs,flag 要从栈里取出，前两者是刚才入栈的，后三者是在调用中断时被入栈的）。这样就把上下文保存好了，紧接着恢复 ss,sp 至内核栈，过程结束。

代码如下：

```
extern _tmpCpuState

save:                                           ; protection
        push eax
        push ds
        mov ax, cs                              ; use OS's ds
        mov ds, ax
        pop ax                                  ; ax=ds
        mov [_tmpCpuState+34], ax
        pop eax
        mov [_tmpCpuState], eax
        mov [_tmpCpuState+4], ebx
        mov [_tmpCpuState+8], ecx
        mov [_tmpCpuState+12], edx
        mov [_tmpCpuState+16], ebp
        mov [_tmpCpuState+20], edi
        mov [_tmpCpuState+24], esi
        pop bx                                  ; bx=ip after 'call save'
        pop ax                                  ; ax=ip
        mov [_tmpCpuState+28], ax
        pop ax                                  ; ax=cs
        mov [_tmpCpuState+30], ax
        pop ax                                  ; ax=flags
        mov [_tmpCpuState+32], ax
        mov [_tmpCpuState+36], es
        mov [_tmpCpuState+38], ss
        mov [_tmpCpuState+40], sp
        mov ax, cs                              ; use OS's stack
        mov ss, ax
        mov sp, [os_sp]
        jmp bx

section .data
        os_sp dw 0
```

其中 os_sp 是内核栈指针的记录。

注意一些细节：call save 时会将 ip 入栈，因此进入 save 后栈顶有两个 ip。

然后是 restart。同理，把 _tmpCpuState 里的东西 mov 到寄存器里即可，需要注意 ds 是最后才被恢复的，在此之前仍要通过 ds 来访问 _tmpCpuState。

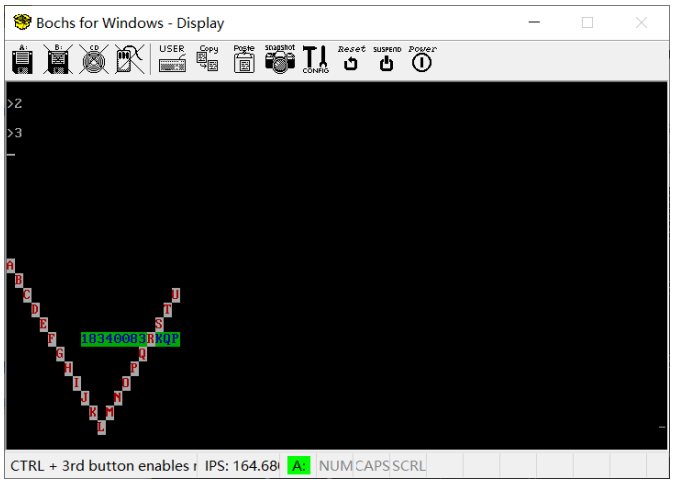
同时，由于需要在 restart 过程里直接返回用户程序，因此在恢复用户栈后，要将 _tmpCpuState 里的 ip,cs,flags 入栈，最后 iret。

代码如下：

```
restart:
        mov eax, [_tmpCpuState]
        mov ebx, [_tmpCpuState+4]
        mov ecx, [_tmpCpuState+8]
        mov edx, [_tmpCpuState+12]
        mov ebp, [_tmpCpuState+16]
```

```
7           mov edi, [_tmpCpuState+20]
8           mov esi, [_tmpCpuState+24]
9           mov [os_sp], sp
10          mov es, [_tmpCpuState+36]
11          mov ss, [_tmpCpuState+38]
12          mov sp, [_tmpCpuState+40]
13          push word [_tmpCpuState+32]             ; flags
14          push word [_tmpCpuState+30]             ; cs
15          push word [_tmpCpuState+28]             ; ip
16          mov ds, [_tmpCpuState+34]
17          iret
```

用实验四的代码（保留风火轮，去掉 OUCH!）进行测试，将 `int 08h` 改为此规范格式，效果如下：（助教可运行 `src\test1\myOS.bxrc` 查看动态效果）



可以看到，风火轮及用户程序都正常运行，不会有数据显示错误，说明这两个过程成功。

5.2 int 20h

该中断的功能为结束用户程序。使用中断方式结束用户程序，优点有三：

- 原本埋在用户程序头部的 `retf` 改成 `int 20h`，避免在用户栈中泄露内核地址；
- 为用户程序编写提供了“强制结束程序”的功能；
- 作为中断处理还可以执行更多代码，为以后的多进程结束处理提供编程位置。

实际上把上次实验的 `end_client` 作为 `int 20h` 就行了：

```
1   _start:
2           ...
3           mov word [gs:0x80], end_client  ; reset int 20h
4           mov word [gs:0x82], cs
```

```
        ...

_load_client:
        ...
        jmp client_seg_addr: 0x100      ; jmp to client

end_client:
        pop ax                          ; pop ip
        pop ax                          ; pop cs
        pop ax                          ; pop flags
        mov ax, cs                      ; recover OS's ds, stack
        mov ds, ax
        mov ss, ax
        mov sp, [os_sp]
        o32 ret
```

5.3 int 21h

该中断的功能为系统调用，例如用户程序的输入、输出，都使用系统调用完成。

此处封装三个系统调用：输出单个字符（功能号 ah=0）、getchar 方式输入单个字符（功能号 ah=1）、getch 方式输入单个字符（功能号 ah=2）。后两者的区别在于：

- getchar 方式有回显，getch 方式无回显；
- 输入缓冲分为键盘设备缓冲、BIOS 在内存中的键盘缓冲、内核自己的输入缓冲，共三级，getchar 从内核输入缓冲中取字符，而 getch 从 BIOS 内存键盘缓冲取字符。（编程实现即可体会差别）

该中断的框架为：

```
server:
        ; cli
        ; simple protection & recover OS's ds, stack
putchar:                                ; ah=0: putchar(al)
        cmp ah, 0
        jnz getchar
                ; process
                jmp end_server
getchar:                                ; ah=1: getchar(al)
        cmp ah, 1
        jnz getch
                ; process
                jmp end_server
getch:                                  ; ah=2: getch(al)
        cmp ah, 2
        jnz end_server
                ; process
                jmp end_server
end_server:
        ; simple recover
        ; sti
        iret
```

要点如下：

1、该中断带有输入参数和返回值，因此不适用 save 和 restart 那种全面现场保护及恢复，需要单独为其编写一个简单的现场保护及恢复功能。

2、在实现多线程以前，必须防止系统再入。举个例子，若在 getchar 执行期间，时间到了触发风火轮，那么风火轮将会执行 save，导致 getchar 期间保存的现场被覆盖、内核栈被重写（因为都从同一个 os_sp 中获得内核栈指针）。简单的 cli 是不够的，因为 BIOS 中断 int 16h 等内有 sti，应将 int 08h 的中断向量暂时恢复，放弃风火轮。

5.3.1 输出单个字符

该功能直接使用 int 10h 实现：（输入参数 al 为 ascii 码）

```
putchar:                                ; ah=0: putchar(al)
        cmp ah, 0
        jnz getchar
                mov bl, 0x07
                mov ah, 0xe
                int 10h
                jmp end_server
```

5.3.2 getchar 方式输入单个字符

之前的内核中，ReadString() 函数已实现了基本的内核输入缓冲功能，之前将缓冲区视为栈，现增加一个头指针而改为队列。

对于 getchar()，逻辑应当是这样的：先检查内核输入缓冲是否存在字符，若存在，则取队首并弹出队首；若不存在，则调用 ReadString() 得到输入缓冲后再取队首并弹出队首。

用 C 实现上述过程：

```
char buf[maxlen];
short buflen,head;
char GetFirstOfBuf()
{
        if (head==buflen) ReadString();
        return buf[head++];
}
```

于是在 int 21h 中只需

```
getchar:                                ; ah=1: getchar(al)
        cmp ah, 1
        jnz getch
                push word 0
                call _GetFirstOfBuf
                jmp end_server
```

返回值一直在 eax 中。

5.3.3 getch 方式输入单个字符

该功能直接使用 int 16h 实现：（返回值 al 为 ascii 码）

```
getch:                                  ; ah=2: getch(al)
        cmp ah, 2
        jnz end_server
                mov ah, 0
                int 16h
                jmp end_server
```

5.4 库设计

现设计一个 IO 库，支持 putchar()、puts()、printf()、getch()、getchar()、gets()、ReadInt() 操作。scanf() 由于其本身格式处理很不通用（标准库中的格式处理也很多 bug），因此不太具有实现价值。

正常 C 库的调用方式为程序头部 include，但考虑到以下原因，改为采用库文件与源程序一起编译链接的方式：

- 开头 include 会使得最终程序开头是别的函数过程而不是主函数 int main，在裸机编程、没有操作系统支持的情况下，无法跳转至主函数开始程序；（也可以通过链接脚本重排 .text 段代码，使得主函数在最前面，但链接脚本的语法太复杂）；
- 库中的一些过程使用汇编和 C 混合编程的方式能增加代码简洁性和效率，但若 include 的话必须使用内嵌汇编。

现在来实现这些过程。putchar()、getch()、getchar() 分别调用三个系统调用即可（在 lib_io.asm 中）。printf() 使用大一时的大作业 myprintf()（其原理为转化好格式后用 putchar() 实现，具体见 src\FinalOS\lib_io.c）。

gets() 的逻辑为，不断调用 getchar()，直至回车：

```
void gets(char *s)
{
        int len=0;
        for(char ch=getchar(); ch!='\r'; ch=getchar()) s[len++]=ch;
        s[len]=0;
}
```

puts() 的逻辑为，不断调用 putchar() 直至输出完成，最后加一个回车换行：

```
void puts(char *s)
{
        int len=strlen(s);
        for(int i=0; i<len; i++) putchar(s[i]);
        putchar('\r'), putchar('\n');
}
```

ReadInt() 为竞赛中的“快速读入”，不断调用 getchar() 读入单位数字，然后拼凑起来：

```
void ReadInt(int *data)
{
        *data=0;
```

```
        char ch=getchar(), neg=0;
        while ((ch<'0' || ch>'9') && ch!='-') ch=getchar();
        if (ch=='-') neg=1, ch=getchar();
        do{
                *data=(*data<<3)+(*data<<1)+ch-'0';
                ch=getchar();
        } while (ch>='0' && ch<='9');
        if (neg==1) *data=-*data;
}
```

5.5 测试程序

现编写一个测试用的 C 用户程序，测试尽可能多的库过程：（src\FinalOS\test.c）

```
int start()
{
        int a,b;
        puts("TEST1: input two number:");
        ReadInt(&a), ReadInt(&b);
        myprintf("the sum is %d\r\n",a+b);

        puts("TEST2: input 5 char in one row:");
        for(int i=0; i<5; i++) putchar(getchar());
        getchar();

        puts("\r\nTEST3: input a string:");
        char s[50];
        gets(s);
        puts(s);

        puts("Finished");
}
```

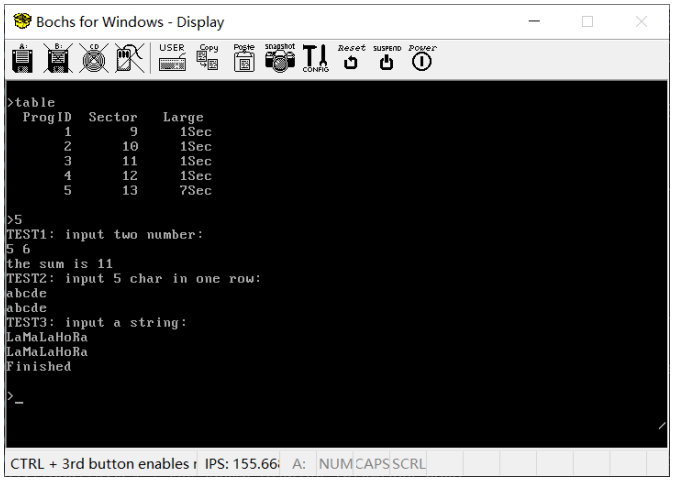
此处没有用常规的 int main()，因为用 main() 函数的话编译器会声明一个 __main 段，而这个段的定义是在操作系统的运行库中，现在编写裸机程序并没有运行库，因此会报错找不到该段。所以我主程序换了一个名字。

编译链接采用源程序、库文件同时编译链接的方式，命令如下：（src\tools\makefile_client.bat，稍有不同）

```
nasm -f elf32 lib_io.asm -o lib_ioASM.o
gcc -c -m32 -march=i386 -mpreferred-stack-boundary=2 -static lib_io.c -o lib_ioC.o
gcc -c -m32 -march=i386 -mpreferred-stack-boundary=2 -static test.c -o test.o
ld --entry=_start -T ld.lds -o test.tmp test.o lib_ioASM.o lib_ioC.o
objcopy -O binary test.tmp test.com
```

安排好顺序使得主程序排在最前面。

测试效果如下：
可以看到，测试程序能正常运行及结束，



6 实验总结

这次实验在技术上没有太多革新，大多是上次实验的中断技术。这次实验的目的在于把操作系统各项服务做全做规范。

6.1 思考的问题

关于操作系统的一些具体实现，在这次实验产生了很多思考。

cmd 要不要独立成用户程序？ 这个问题是在编写 IO 库时想要复用代码时想到的，这个问题的本质是考虑到了 cmd 独立成用户程序的利弊。

利在于，第一，cmd 本身需要使用功能强大的 IO 操作，该操作已在用户库文件中详细实现，若将其存于内核，则又要在内核中重复实现一次，造成浪费；第二，将来若涉及多进程、用户程序后台执行，那么 cmd 本身需要当作独立进程来处理，若仍为内核部分，则容易导致系统再入。

弊在于，第一，cmd 涉及很多内核高权限操作，独立成为用户程序将造成很多不便；第二，若当作独立进程，则在用户程序后台执行和不后台执行时，cmd 的调度优先权会频繁改变，较为繁琐。

内核要不要 IO？ 老师在实验课上说内核不要有 IO，仔细思考后我认为不妥，若用户程序意外崩溃，应当由内核执行中断时输出一些崩溃信息，此处至少有“字符串输出”级别的 IO 操作。

输入缓冲到底怎么解决？ 这是在探究 `getchar()` 和 `getch()` 的区别时出现的问题，后来通过实验理清了 5.3 节所述的三级缓冲结构，于是想到了用之前实现的 `ReadString()` 作为内核输入缓冲，再设计 `GetFirstOfBuf()` 就实现了标准的 `getchar()`。

系统再入怎么办？ 一开始大家都坚定不移地认为中断嵌套没问题，但实际操作起来才发现，系统再入真的会发生很多问题，包括系统栈会覆盖重写。这是由于系统再入的本质是多线程，本就应当需要两份 CPU 上下文、栈空间。换句话说，在实现多线程以前，系统服务中断时都不能转风火轮了。当然其他的嵌套也不可以了。

6.2 试错细节

一些常见的错误包括：

1、bochs 的 enter 键输入只产生 `\r`，不产生 `\n`；

2、本实验中 C 代码均要加上 `code16gcc`，使语句加上反转前缀，否则在虚拟机中解析指令会不准确。C 代码在栈对齐等方面永远是 32 位起步的，只有内存寻址可以做到实模式，因此强行按 16 位解释 C 程序是会错误的；

3、C 代码的用户程序在最后 `ret` 时会从栈中弹出 4 个字节，因此内核在加载用户程序前必须 `push` 足够多的 0，以保证 `ret` 不会出错。

参考文献

[1] minix.asm

[2] 田宇,一个64位操作系统的设计与实现,人民邮电出版社