

实验五：实现系统调用

计算机科学与技术 18340146 宋渝杰

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

实验要求：

1. 了解PC系统的软中断指令的原理
2. 掌握x86汇编语言软中断的响应处理编程方法
3. 扩展实验四的内核程序，增加输入输出服务的系统调用。
4. C语言的库设计，实现putch()、getch()、printf()、scanf()等基本输入输出库过程
5. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验内容：

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和int 22h软中断的处理程序，其中，int 20h用于用户程序结束是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；int22h功能未定，先实现为屏幕某处显示INT22H
3. 保留无敌风火轮显示，取消触碰键盘显示OUCH!这样功能
4. 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，汇编产生libs.obj
5. 利用自己设计的C库libs.obj，编写一个使用这些库函数的C语言用户程序，再编译,在与libs.obj一起链接，产生COM程序。增加内核命令执行这个程序
6. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验过程：

操作环境：

- 操作系统：win10 + Linux
- 虚拟机：VirtualBox
- C 编译器：gcc
- x86 编译器：nasm
- 链接器：ld

步骤一：实现 `save()` 和 `restart()`

实现 `save()`：

在中断发生时，实模式下的 CPU 会将 flags、cs、ip 先后压入当前被中断程序（时间中断时为内核，系统调用时为用户程序）的堆栈中，之后跳转到中断处理函数中进行处理。

包括上述三个寄存器，还有以下寄存器需要保存：

- 段寄存器：es, ds, ss
- 主寄存器：di, si, bp, sp, ax, bx, cx, dx

因此，我在 c 模块声明结构体 PCB 用户存储各寄存器信息，并在汇编模块利用写地址的方式，将寄存器的数值存入 c 模块相应位置中。需要提出的是，汇编模块我先利用了 si 指向 c 模块结构体 PCB 的地址，之后用地址偏移写入寄存器内容，因此先暂存 si 寄存器的值，在写完所有内容之后，最后保存 si 寄存器的值。

更具体的每个步骤的解释参见下文代码和注释：

c 模块：

```
struct PCB {  
    // 由于是利用地址偏移保存，因此下面变量的顺序可以随意  
    int ip, cs, es, ds, ss, di, si, bp, sp, ax, bx, cx, dx, flags, pid;  
} q[5], *qi = q; // 利用指针写地址，因此 q 需声明为数组形式
```

汇编模块：

```
save:  
    push ds  
    push cs  
    pop ds                ; ds 指向内核  
    pop word[save_ds]     ; 保存了原始的 ds，即用户程序  
    pop word[save_cs]     ; 保存了 save 返回的地址  
    mov word[save_si],si  ; 暂存 si  
    mov si,word[qi]       ; c 中的 save 结构体  
    pop word[si]          ; ip  
    pop word[si+4]        ; cs  
    pop word[si+8]        ; flags  
    mov word[si+12],es    ; es  
    push word[save_ds]  
    pop word[si+16]       ; ds  
    mov word[si+20],ss    ; ss  
    mov word[si+24],ax    ; ax  
    mov word[si+28],bx    ; bx  
    mov word[si+32],cx    ; cx  
    mov word[si+36],dx    ; dx  
    mov word[si+40],di    ; di  
    mov word[si+44],bp    ; bp  
    mov word[si+48],sp    ; sp  
    push word[save_si]  
    pop word[si+52]       ; si  
    jmp word[save_cs]     ; 跳转回 call save 下一步  
  
datadef:  
    ; 这些是中间变量，不保存实际信息  
    save_cs dw 0  
    save_si dw 0  
    save_ds dw 0  
    save_pid dw 0
```

实现 restart():

restart 过程实际上是 save 的逆过程，因此只需把 save 保存的所有寄存器的值读出来，放入对应的位置，最后使用 iret 将 flags、cs、ip 反向出栈存入对应寄存器即可。和 save 一样，我同样先使用 si 指向 c 模块结构体的位置，用地址偏移读出寄存器的值并存入后，最后再 restart si。

汇编模块：

```
restart:
    mov si,[qi]
    mov es,word[si+12]      ; es
    mov ss,word[si+20]      ; ss
    mov ax,word[si+24]      ; ax
    mov bx,word[si+28]      ; bx
    mov cx,word[si+32]      ; cx
    mov dx,word[si+36]      ; dx
    mov di,word[si+40]      ; di
    mov bp,word[si+44]      ; bp
    mov sp,word[si+48]      ; sp
    push word[si+8]         ; flags
    push word[si+4]         ; cs
    push word[si]           ; ip
    push word[si+52]
    push word[si+16]
    pop ds                  ; ds
    pop si                  ; si
    iret                   ; 中断结束，返回 cs:ip 处执行
```

最后，修改 int 8h 和原有的 int 20h，调用 save() 和 restart()

```
int20h:
    call save
    call cls                ; 用户程序退出后清屏
    jmp main
    jmp restart             ; 结构对称，实际上该代码不执行

int8h:
    cli                    ; 屏蔽外部中断
    call save
    push 0
    call draw               ; c 模块“数字钟”显示
    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al             ; 中断结束
    pop ax
    sti                    ; 解除屏蔽
    jmp restart
```

编译命令行如下：和实验四基本一样，注意修改文件名即可，因此不再赘述：

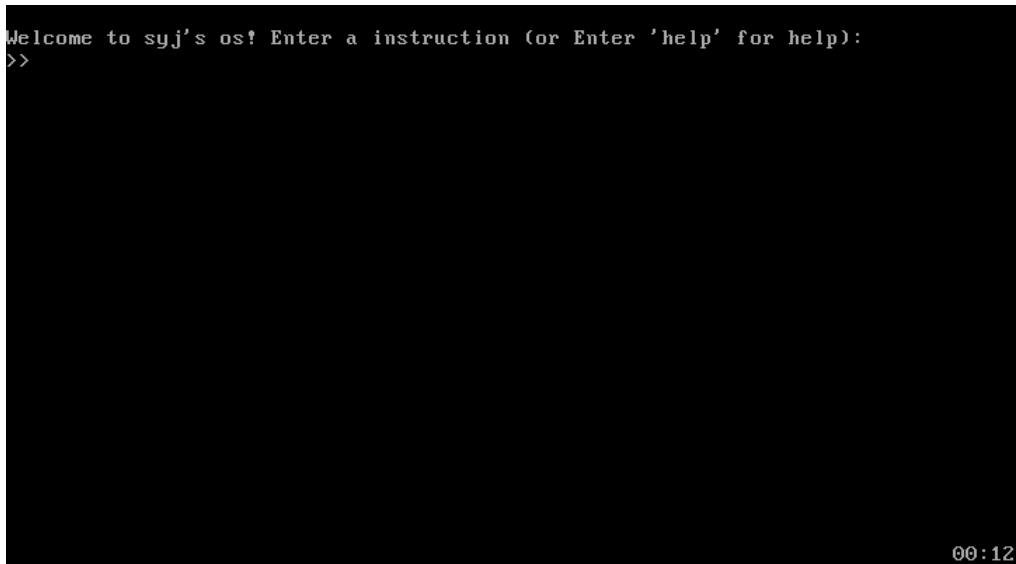
```
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 5-1c.c -o 5-1c.o
nasm -felf 5-1asm.asm -o 5-1asm.o
ld -m elf_i386 -N --oformat binary -Ttext 0x7e00 5-1asm.o 5-1c.o -o 5-1.bin
nasm 5-1os.asm -o 5-1os.bin

rm -f syjos.img
/sbin/mkfs.msdos -C syjos.img 1440
dd if=1.com of=syjos.img seek=18 conv=notrunc
dd if=2.com of=syjos.img seek=19 conv=notrunc
dd if=3.com of=syjos.img seek=20 conv=notrunc
dd if=4.com of=syjos.img seek=21 conv=notrunc
dd if=5-1.bin of=syjos.img seek=1 conv=notrunc
dd if=5-1os.bin of=syjos.img conv=notrunc
```

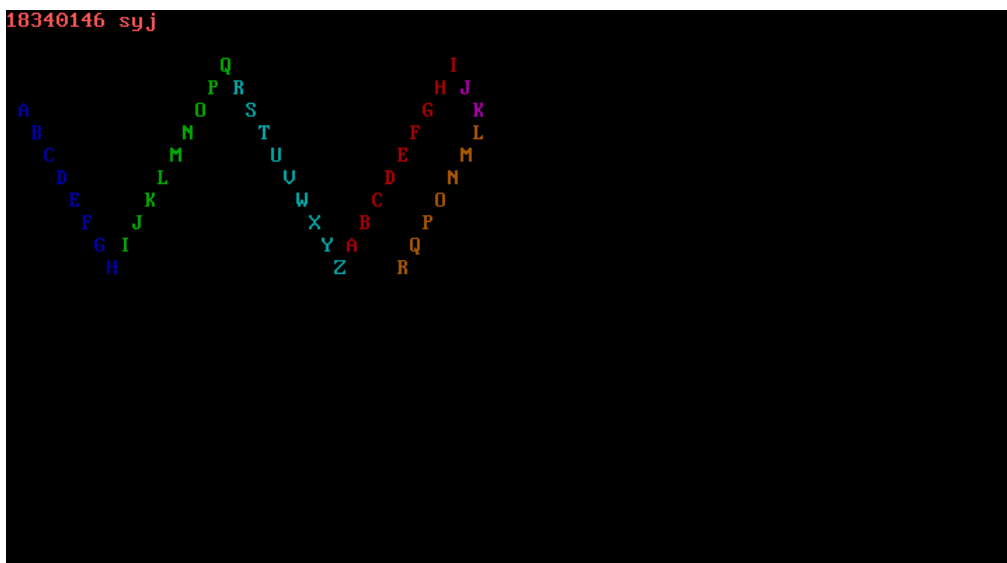
测试 save() 和 restart():

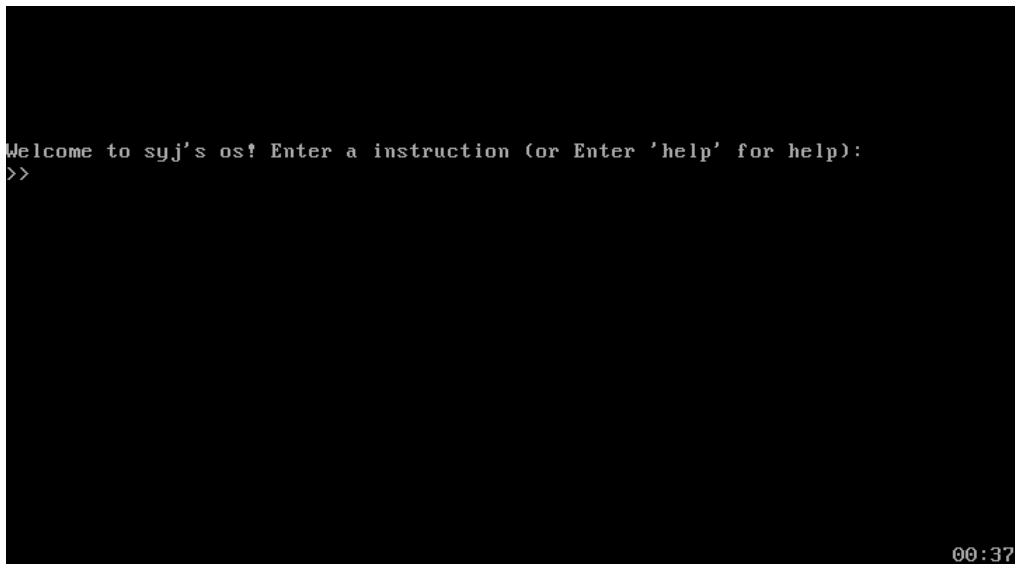
生成 5-1.img，测试结果如下图：

内核刚运行时，可以看出使用了 save() 和 restart() 过程的时钟中断正常运行，数字钟正常显示：



打开用户程序 1，用户程序通过 int 20h 中断返回，可以看出返回正常：





步骤二：内核增加 `int 20h`、`int 21h` 和 `int 22h`

增加 `int 20h`:

在我的实验二中，就已经实现了 `int 20h`，用于用户程序结束时返回内核，准备接受新的命令。

实验二中对 `int 20h` 的实现过程和思路如下图：（注意，下方小段文字为图片）

3. 添加 `int 20h` 中断，实现从用户程序返回监控程序的功能：由于前面讲到用户程序显示 100 个字符后会自动停止，因此 `int 20h` 只需直接返回监控程序顶部即可

在此基础上把步骤一的 `save()` 和 `restart()` 加入后，最终代码如下：

```
int20h:
    call save
    call cls                ; 用户程序退出后清屏
    jmp main
    jmp restart            ; 结构对称，实际上该代码不执行
```

增加 `int 21h`:

`int 21h` 用于系统调用，并实现几个简单系统调用功能。在本次实验五中，我实现了 7 个系统调用功能，具体信息如下表：

| 功能号 | 入口参数 | 出口参数 | 功能描述 |
|-----|-------------------------------------|---------|----------------------------------|
| 1 | ah = 1, es:dx 为字符串地址 | 无 | 将 es:dx 位置的一个字符串转化成大写 |
| 2 | ah = 2, es:dx 为字符串地址 | 无 | 将 es:dx 位置的一个字符串转化成小写 |
| 3 | ah = 3, es:dx 为字符串地址, bx = 翻转的字符串长度 | 无 | 将 es:dx 位置的一个字符串前 bx 位翻转 |
| 4 | ah = 4, es:dx 为字符串地址 | ax = 数值 | 将 es:dx 位置的一个 10 进制字符串转变为对应的数值 |
| 5 | ah = 5, es:dx 为字符串地址, bx = 要转换的数值 | 无 | 将 bx 的数值转变为 es:dx 位置的一个 10 进制字符串 |
| 6 | ah = 6, es:dx 为字符串地址 | ax = 数值 | 将 es:dx 位置的一个 2 进制字符串转变为对应的数值 |
| 7 | ah = 7, es:dx 为字符串地址, bx = 要转换的数值 | 无 | 将 bx 的数值转变为 es:dx 位置的一个 2 进制字符串 |

下面是各个系统调用功能的实现思路和步骤：

功能号 ah=01h 的系统调用（字符串转大写）：

实现的思路与实验 3-2 相似：汇编模块将参数 es、dx 压栈传递给 c 模块函数，在 c 中对字符串进行转大写操作，最后无需返回值，指针所指的字符串已完成转换。

c 模块：

```
void toupper(char* s) {
    int i = 0;
    while (s[i]) {
        if (s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i]-32; // 转大写
        i++;
    }
}
```

汇编模块：

```
sys_1:
    push es                ; 传递参数
    push dx                ; 传递参数
    call dword toupper
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

功能号 ah=02h 的系统调用（字符串转小写）：

同理，在上面的基础上，c 模块函数大写改成小写即可

c 模块:

```
void tolower(char* s) {
    int i = 0;
    while (s[i]) {
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] = s[i]+32; // 转小写
        i++;
    }
}
```

汇编模块:

```
sys_2:
    push es                ; 传递参数
    push dx                ; 传递参数
    call dword tolower
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

功能号 ah=03h 的系统调用 (字符串反转) :

这里与上文会有一些不同: c 模块函数多接收一个参数, 表示翻转的字符串长度, 因此汇编模块参数压栈和多余参数出栈的步骤要多一点

c 模块:

```
void reverse(char* s, int len) {
    int l=0,r=len-1;
    char temp;
    for (; l<r; l++, r--) { // 翻转
        temp = *(s+l);
        *(s+l) = *(s+r);
        *(s+r) = temp;
    }
}
```

汇编模块:

```
sys_3:
    mov ax, 0
    push ax                ; 传递参数 len
    push bx                ; 传递参数 len
    push es                ; 传递参数 s
    push dx                ; 传递参数 s
    call dword reverse
    pop bx                 ; 丢弃参数
    pop ax                 ; 丢弃参数
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

功能号 ah=04h 的系统调用 (十进制数字字符串转换成 int 类型数值) :

这里与上文也有一些不同：c 模块函数多了一个返回值，会自动赋值到寄存器 AX 上，因此后文想要获取这个返回值的话，去寄存器 AX 就可以找到

c 模块：

```
int atoi(char *s) {
    int ans = 0, i;
    for (i=0; s[i]!='\0'; ++i)
        ans = ans*10+s[i]-48;
    return ans;
}
```

汇编模块：

```
sys_4:
    push es                ; 传递参数
    push dx                ; 传递参数
    call dword atoi
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

功能号 ah=05h 的系统调用（int 类型数值转化成十进制字符串）：

这里 c 模块函数有两个参数，和 ah=03h 的系统调用 c 模块函数类似，但是参数位置不同，汇编模块压栈时记得交换一下参数压栈次序即可。

c 模块：

```
void itoa(int num, char* str) {
    int i = 0;
    if (num == 0) // 特判 num = 0
        str[i++] = '0';
    while (num) {
        str[i++] = num%10+48;
        num /= 10;
    }
    str[i] = '\0';
    reverse(str, i); // 别忘了翻转
}
```

汇编模块：

```
sys_5:
    push es                ; 传递参数 s
    push dx                ; 传递参数 s
    mov ax, 0
    push ax                ; 传递参数 val
    push bx                ; 传递参数 val
    call dword itoa
    pop bx                 ; 丢弃参数
    pop ax                 ; 丢弃参数
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```


功能号 ah=06h 的系统调用（二进制数字字符串转换成 int 类型数值）：

这里与 ah=04h 十进制字符串转成 int 类似，代码也是把 *10 改成 *2 即可

c 模块：

```
int atoi_2(char *s) {
    int ans = 0, i;
    for (i=0; s[i]!='\0'; ++i)
        ans = ans*2+s[i]-48;
    return ans;
}
```

汇编模块：

```
sys_6:
    push es                ; 传递参数
    push dx                ; 传递参数
    call dword atoi_2
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

功能号 ah=07h 的系统调用（int 类型数值转化成二进制字符串）：

这里与 ah=05h int 转十进制字符串类似，代码也是把 10 改成 2 即可

c 模块：

```
void itoa(int num, char* str) {
    int i = 0;
    if (num == 0) // 特判 num = 0
        str[i++] = '0';
    while (num) {
        str[i++] = num%2+48;
        num /= 2;
    }
    str[i] = '\0';
    reverse(str, i); // 别忘了翻转
}
```

汇编模块：

```
sys_7:
    push es                ; 传递参数 s
    push dx                ; 传递参数 s
    mov ax, 0
    push ax                ; 传递参数 val
    push bx                ; 传递参数 val
    call dword itoa_2
    pop bx                 ; 丢弃参数
    pop ax                 ; 丢弃参数
    pop dx                 ; 丢弃参数
    pop es                 ; 丢弃参数
    ret
```

将 7 个系统调用整合成 int 21h:

上文汇编模块定义了 7 个系统调用函数，这里需要将它们整合到 21 号中断函数中，根据功能号 AX 的具体值调用相应的函数即可，同时记得写入中断向量表。

汇编模块：

```
extern toupper,tolower,reverse,atoi,itoa,atoi_2,itoa_2

    mov ax, 0000h                ; 内存前64k放置的中断向量表，将段寄存器指向该处
    mov es, ax
    mov ax, 21h                  ; 定义 21 号中断：系统调用
    mov bx, 4
    mul bx
    mov si, ax
    mov ax, int21h                ; 写入函数 int21h
    mov [es:si], ax
    add si, 2
    mov ax, cs
    mov [es:si], ax

int21h:
    call save                    ; save
    push ds
    push si                      ; 用si作为内部临时寄存器
    mov si, cs
    mov ds, si                  ; ds = cs
    mov si, ax
    shr si, 8                   ; si = 功能号
    add si, si                   ; si = 2 * 功能号
    sub si, 2
    call [sys_table+si]         ; 系统调用函数
    pop si
    pop ds
    jmp restart                 ; restart
sys_table:                      ; 存放功能号与系统调用函数映射的表
    dw sys_1,sys_2,sys_3,sys_4,sys_5,sys_6,sys_7
```

增加 int 22h:

实验要求 int 22h 功能未定，先实现为屏幕某处显示 INT22H。因此我也只是基本实现该步骤，并不对其进行调用，在后续的实验中如果还有对 int 22h 的操作步骤，再对其进行调用

```
int22h:
    call save
    mov ax, ouch
    mov bp, ax
    mov cx, len
    mov ax, 1301h
    mov bx, 000fh
    mov dx, 0505h
    int 10h                    ; 输出 INT22H
    jmp restart

    ouch db 'INT22H', 0
    len equ $-ouch
```

编写 int 21h 测试文件:

int 20h 在实验二以及这次实验的步骤一中已经测试, int 22h 尚未调用, 因此这次步骤二的测试部分只需测试 int 21h 的功能即可

编写用户程序 5.asm, 用于测试系统调用的各项功能:

测试 ah=01h, ah=02h, ah=03h: 定义一个名为 test1 的字符串变量, 包含着多个大写字母和小写字母, 先转换成大写, 再转换成小写, 再将前 7 位字符翻转:

```
mov ax, cs
mov es, ax                ; es=cs
mov dx, test1             ; es:dx=串地址
mov ah, 01h               ; 系统调用功能号ah=01h, 大写转小写
int 21h
...                        ; 输出提示信息
mov ax, cs
mov es, ax                ; es=cs
mov dx, test1             ; es:dx=串地址
mov ah, 02h               ; 系统调用功能号ah=02h, 小写转大写
int 21h
...                        ; 输出提示信息
mov ax, cs
mov es, ax                ; es=cs
mov dx, test1             ; es:dx=串地址
mov bx, 7                  ; 部分反转, 只反转前 7 位
mov ah, 03h               ; 系统调用功能号ah=03h, 反转
int 21h
...                        ; 输出提示信息
test1 db 'AbCdEfGsyj', 0
len1 equ $-test1
```

测试 ah=04h, ah=05h: 定义一个名为 test2 的字符串变量, 用于测试系统调用 4 号功能; 同时给 BX 赋值 16388, 用于测试系统调用 5 号功能

需要特别注意的是, 上文提到 4 号调用的返回值返回到 AX 中, 但是中断结束调用 restart() 后, AX 的值将会被覆盖, 因此这种情况下无法检测 4 号调用是否正确 (下文将用别的方式测试)

```
mov ax, cs
mov es, ax                ; es=cs
mov dx, test2             ; es:dx=串地址
mov ah, 04h               ; 系统调用功能号ah=04h, atoi
int 21h                   ; ax=转换后的数字

mov bx, 16388
mov ax, cs
mov es, ax                ; es=cs
mov dx, test2             ; es:dx=串地址
mov ah, 05h               ; 系统调用功能号ah=05h, itoa
int 21h                   ; es:dx=转换后的数字字符串
...                        ; 输出提示信息
test2 db '16388', 0
len2 equ $-test2
```

测试 ah=06h, ah=07h: 定义一个名为 test3 的字符串变量, 用于测试系统调用 6 号功能; 同时给 BX 赋值 46, 用于测试系统调用 7 号功能

同样地，这种条件下无法测试 6 号调用是否正确，下文也会采取别的方式测试

```
mov ax, cs
mov es, ax                ; es=cs
mov dx, test3             ; es:dx=串地址
mov ah, 06h               ; 系统调用功能号ah=06h, atoi_2
int 21h                   ; ax=转换后的数字

mov bx, 46
mov ax, cs
mov es, ax                ; es=cs
mov dx, test3             ; es:dx=串地址
mov ah, 07h               ; 系统调用功能号ah=07h, itoa_2
int 21h                   ; es:dx=转换后的数字字符串
...                       ; 输出提示信息
test3 db '101110', 0
len3 equ $-test3
```

之后，修改一下内核的提示信息，加入用户程序 5 的信息：

```
static char exestr[] =
    "exe1      -- LeftUp      512 bytes\n"
    "exe2      -- RightUp     512 bytes\n"
    "exe3      -- LeftDown    512 bytes\n"
    "exe4      -- RightDown   512 bytes\n"
    "exe5      -- RightDown   401 bytes\n"; // 只需添加这行即可
```

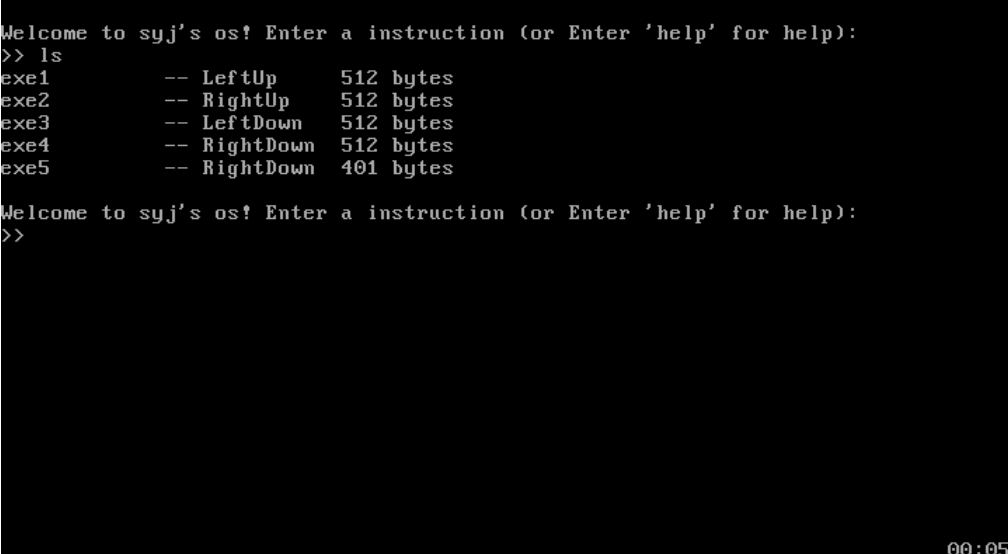
最后，在上文的编译命令行添加以下代码，将 5.asm 生成 5.com 并烧入 img 文件内：

```
nasm 5.asm -o 5.com
dd if=5.com of=syjos.img seek=22 conv=notrunc
```

测试 int 21h:

生成 5-2.img，测试结果如下图：

输入指令 ls，可以显示新的用户程序信息：



```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> ls
exe1      -- LeftUp      512 bytes
exe2      -- RightUp     512 bytes
exe3      -- LeftDown    512 bytes
exe4      -- RightDown   512 bytes
exe5      -- RightDown   401 bytes

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>
```

输入指令 exe，再输入 5，运行测试系统调用的用户程序：

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> ls
exe1      -- LeftUp      512 bytes
exe2      -- RightUp    512 bytes
exe3      -- LeftDown   512 bytes
exe4      -- RightDown  512 bytes
exe5      -- RightDown  401 bytes

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exe
Enter a number (1-5) to open an exe:

>> _
```

01:47

测试结果如下图，具体解释如下：一开始系统调用 1 号功能处理字符串 `AbCdEfGSyJ`，将字符串转成大写，生成新的字符串 `ABCDEFGSYJ`；系统调用 2 号功能处理新字符串，将字符串转成小写，生成 `abcdefghijkl`；系统调用 3 号功能处理新字符串，将字符串前 7 位翻转，生成 `gfedcbasyj`；系统调用 4 号功能处理字符串 `16388`，但无法验证正确性；系统调用 5 号功能处理数值 16388，生成 10 进制字符串 `16388`；系统调用 6 号功能处理字符串 `101110`，但无法验证正确性；系统调用 7 号功能处理数值 46，生成 2 进制字符串 `101110`。可以看出 1、2、3、5、7 号系统调用均能正确运行。

```
syscall_1  ABCDEFGSYJ
syscall_2  abcdefgsyJ
syscall_3  gfedcbasyj
syscall_5  16388
syscall_7  101110
Enter any to return
```

下面验证 4、6 号系统调用的正确性：先去掉 `save()` 和 `restart()` 函数，保证寄存器 AX 内保存到系统调用返回值，之后把上文中测试 4、6 号系统调用的代码 `mov bx,16388` 和 `mov bx,46` 都修改成 `mov bx,ax`，将 4、6 号系统调用的返回值成为 5、7 号系统调用的参数。

修改代码之后的测试结果如下图（与上图相同），解释如下：因为 4、6 号系统调用分别处理字符串 `16388` 和 `101110`，返回值分别为 int 类型数值 16388 和 46，再通过 5、7 号系统调用处理这两个数值，生成的新字符串为 `16388` 和 `101110`，与图中显示相符。

```
syscall_1  ABCDEFGSYJ
syscall_2  abcdefgsyj
syscall_3  gfedcbasyj
syscall_5  16388
syscall_7  101110

Enter any to return
```

步骤三：保留无敌风火轮显示，取消触碰键盘显示 OUCH! 这样功能

这部分比较简单，保留实验四的时钟中断（我把风火轮升级为了数字钟），并把我实验四中的 `draw()` 函数关于触碰键盘显示 OUCH! 的代码直接删掉即可。

```
void draw() {
    if (judge == 0) { // 数字钟部分，保留即可
        count++;
        if (count >= 18) {
            count = 0;
            num1++;
            if (num1 >= 10) {
                num1 = 0;
                num2++;
                if (num2 >= 6) {
                    num2 = 0;
                    num3++;
                    if (num3 >= 10) {
                        num3 = 0;
                        num4++;
                        if (num4 >= 6) num4 = 0;
                    }
                }
            }
        }
        pchar2((char)(num4+48), 24, 75);
        pchar2((char)(num3+48), 24, 76);
        pchar2(':', 24, 77);
        pchar2((char)(num2+48), 24, 78);
        pchar2((char)(num1+48), 24, 79);
    }
}

/* 下面为触碰键盘显示 OUCH! 的代码，直接删掉即可
else {
    schar();
    if (in == 1) {
        pchar2('O', xc, yc);
        pchar2('U', xc, yc+1);
        pchar2('C', xc, yc+2);
        pchar2('H', xc, yc+3);
    }
}
```

```

        pchar2('!',xc,yc+4);
        pchar2('o',xc,yc+5);
        pchar2('u',xc,yc+6);
        pchar2('c',xc,yc+7);
        pchar2('H',xc,yc+8);
        pchar2('!',xc,yc+9);
        xc = (xc+2)%24;
        yc = (yc+18)%65;
        in = 0;
    }
}
*/
}

```

步骤四：进行 c 语言的库设计

在我的实验三中，已经实现了一些 c 语言的输入输出函数，代码如下：

```

int x = 0, y = 0;
void pchar(char c) { // 输出一个字符
    if (c != '\r' && c != '\n') {
        asm volatile( // asm 汇编内嵌编程，在屏幕上显示字符
            "push es\n"
            "mov es, ax\n"
            "mov es:[bx],cx\n"
            "pop es\n"
            :
            : "a"(0xB800), "b"((x*80+y)*2), "c"((0x07<<8)+c)
            :); // a: 显存起始地址 b: 行列位置 c: 颜色和字符
        if (++y >= 80) {
            if (++x >= 25)
                x = 0;
            y = 0;
        }
        asm volatile( // asm 汇编内嵌编程，设置光标位置
            "int 0x10\n"
            :
            : "a"(0x0200), "b"(0), "d"((x<<8)|y)); // a: 功能号 b: 页码 c: 位置
        return;
    }
}
do
    pchar(' ');
while (y);

char gchar() { // 输入一个字符
    char ch;
    asm volatile("int 0x16\n" // asm 汇编内嵌编程，从键盘获取一个字符
        : "a"(ch) // 获取的字符放入 ch 中
        : "a"(0x1000));
    return ch;
}

void pstr(char *s) { // 输出一个字符串
    for (; *s; ++s)

```

```

        pchar(*s);                // 通过循环 pchar 来实现输出字符串
    }

    void gstr(char *s) { // 输入一个字符串
        for (; ++s) {
            pchar(*s = gchar());    // 循环 gchar 并通过 pchar 显示输入的字符
            if (*s == '\r' || *s == '\n')
                break;
        }
        *s = '\0';
    }
}

```

目前的输入输出函数已经可以实现输入、输出一个字符和一个字符串。在这次实验中，新添加输入、输出一个 int 类型数值的 io 函数：

```

void pint(int n) { // 输出一个 int 数值
    char str[10];
    int i = 0;
    while (n) {
        str[i++] = n%10+48;
        n /= 10;
    }
    for (; i>0; i--)
        pchar(str[i-1]);
}

int gint() { // 输入一个 int 数值
    int n = 0;
    char c;
    while (1) {
        c = gchar();
        if (c == '\r' || c == '\n')
            break;
        while (c > '9' || c < '0') // 防止错误输入
            c = gchar();
        pchar(c);
        n = n*10+c-48;
    }
    return n;
}

```

将上述代码独立出来，生成 io.c 库文件，之后在 win 下输入编译命令行 `gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c io.c -o io.o` 生成 io.o 文件，用于之后 c 用户程序的链接

需要注意的是，这里的 io 库设计并没有像老师报告例子中的调用内核系统中断来实现 io，而是采用 asm 内嵌汇编形式写成 io.c 库文件，主要有以下考虑：

- 实现的效果会完全一样
- 在实验三中已在内核用 asm 内嵌汇编形式封装了 io 函数，直接取出函数生成 io 库会节省许多不必要的时间浪费
- 如果用系统调用方式封装 io 函数，会有很大的参数和返回值的传递问题：用户程序调用 io 函数 -> 传参给系统调用 -> 系统调用接收参数，内核汇编模块使用代码进行 io 操作（或者更麻烦的：再传参给内核 c 模块进行 io 操作，再传返回值给内核汇编模块） -> io 操作返回值传递给系统调用 -> 系统调用将返回值传递给用户程序的 io 函数，参数传递十分复杂且混乱，操作难度很大

- `restart()` 问题：上文提到系统调用方式函数返回值会赋给 AX 寄存器，而当 `restart()` 之后，该返回值将会被覆盖，无法返回给用户程序 `io` 函数

至此，已经完成了基本的 c 语言 `io` 库设计

步骤五：新增用户程序，用于测试 c 语言 `io` 库

这部分也相对简单，先编写 c 测试程序如下：

```
extern void pchar(char); // 声明库函数
extern char gchar();
extern void pstr(char*);
extern void gstr(char*);
extern void pint(int);
extern int gint();

void cmain() {
    char msg[] = "Please input a char: \n\n>> ";
    char* ptr = msg;
    pstr(ptr);
    char c = gchar(); // 测试读入一个字符
    pchar(c);
    char msg2[] = "\n\nThe input char is: ";
    ptr = msg2;
    pstr(ptr);
    pchar(c); // 测试输出一个字符
    char msg3[] = "\n\nPlease input a string: \n\n>> ";
    ptr = msg3;
    pstr(ptr);
    char s[100];
    gstr(s); // 测试读入一个字符串
    char msg4[] = "\n\nThe input string is: ";
    ptr = msg4;
    pstr(ptr);
    ptr = s;
    pstr(ptr); // 测试输出一个字符串
    char msg5[] = "\n\nPlease input an int: \n\n>> ";
    ptr = msg5;
    pstr(ptr);
    int n = gint(); // 测试读入一个 int 数值
    char msg6[] = "\n\nThe input int is: ";
    ptr = msg6;
    pstr(ptr);
    pint(n); // 测试输出一个 int 数值
    char msg7[] = "\n\n          Enter any to return";
    ptr = msg7;
    pstr(ptr);
    return;
}
```

需要注意的是，直接使用 `ld` 将 c 测试程序和 c 库链接生成 `6.com` 用户程序会出现问题，找不到进入函数，我的解决方式是：像实验三步骤二一样，编写汇编程序辅助进入用户程序：

```

bits 16
extern cmain
start:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
    call dword cmain                ; 直接 call c 主函数即可
end:
    mov ah, 0
    int 16h
    int 20h                        ; 操作完之后, 按键返回内核程序

```

之后, 修改一下内核的提示信息: 因为我们加入了用户程序 6.com, 因此在提示信息中加入它的信息

```

static char exestr[] =
    "exe1      -- LeftUp      512 bytes\n"
    "exe2      -- RightUp     512 bytes\n"
    "exe3      -- LeftDown    512 bytes\n"
    "exe4      -- RightDown   512 bytes\n"
    "exe5      -- RightDown   401 bytes\n"
    "exe6      -- RightDown   1724 bytes\n"; // 加入这一行即可

```

需要注意的是, 用户程序 6.com 已经超过了 512 bytes, 即一个扇区, 这里我们需要把内核汇编部分的加载用户程序扇区数将 1 调大, 超过用户程序最大值即可, 这里调整成 5:

```

LoadnEx:
    push ebp                ; ebp入栈
    mov ebp, esp            ; 因为esp是堆栈指针, 无法暂借使用, 所以得用ebp来存取堆栈
    mov ecx, [ebp+8]
    mov ax, cs              ; 段地址: 存放数据的内存基地址
    mov es, ax              ; 设置段地址
    mov bx, OffSetOfUserPrg1 ; 偏移地址; 存放数据的内存偏移地址
    mov ah, 2               ; 功能号
    mov al, 5               ; 扇区数 (调整这里为 5 )
    mov dl, 0               ; 功能号
    mov dh, 1               ; 磁头号: 起始编号为0
    mov ch, 0               ; 柱面号: 起始编号为0
    int 13h                 ; BIOS的13h功能
    call cls                 ; 调用清屏函数
    jmp OffSetOfUserPrg1
    mov esp, ebp
    pop ebp
    ret

```

Linux 下使用下列编译命令行整合成用户程序 6.com, 并烧入 5-5.img 中, 参数和之前的编译命令行一样, 这里不再赘述

```

gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 6.c -o 6c.o
nasm -felf 6.asm -o 6asm.o
ld -m elf_i386 -N --oformat binary -Ttext 0xa100 6asm.o 6c.o io.o -o 6.com
dd if=6.com of=syjos.img seek=23 conv=notrunc

```

测试 io 库:

生成完整的 5-5.img, 测试效果如下:

进入内核后输入指令 ls, 可以发现输出信息中包含了最新的用户程序 6:

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> ls
exe1      -- LeftUp      512 bytes
exe2      -- RightUp     512 bytes
exe3      -- LeftDown   512 bytes
exe4      -- RightDown  512 bytes
exe5      -- RightDown  401 bytes
exe6      -- RightDown  1724 bytes

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>
```

00:02

输入指令 exe, 再输入 6, 进入 c 函数 io 库的测试用户程序:

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> ls
exe1      -- LeftUp      512 bytes
exe2      -- RightUp     512 bytes
exe3      -- LeftDown   512 bytes
exe4      -- RightDown  512 bytes
exe5      -- RightDown  401 bytes
exe6      -- RightDown  1724 bytes

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exe
Enter a number (1-6) to open an exe:
>>
```

00:51

进入测试程序后, 按照提示分别输入 char 字符 s, string 字符串 hello (回车), int 数值 16388 (回车), 程序相应输出你输入的字符、字符串和 int 数值, 可以看见 io 库函数正常运行

```
Please input a char:
>> s

The input char is: s

Please input a string:
>> hello

The input string is: hello

Please input an int:
>> 16388

The input int is: 16388

Enter any to return
```

实验心得：

这次实验难度和实验三类似（即很难），研究的新内容很多，有 `save()` 和 `restart()` 保存和重启过程、系统调用、c 语言 io 库的编写，现有的参考资料也很少，导致我也花费了不少时间去探索和研究

而对于其它技术层面的心得，我也在此一并讲述：

- **代码纠错：**这次程序在步骤一 `save()` 和 `restart()` 的代码看了老师给出的代码（旧版），但完全没有套用，由我自己理解且重新编写，其它部分均未参考老师代码，因此不存在老师代码纠错部分
- **实验过程中出现的出错问题及解决方式总结：**一个是 `save()` 和 `restart()` 数据存储和重载过程，采取了很多方式去尝试，老师的代码（实验六的 ppt）是采取数据压栈出栈的方式，个人不太喜欢，因为会产生 sp 指针偏移，需要调整以指向正确位置，之后我采取了汇编模块声明十几个变量，分别存储对应的寄存器值，这种方式对一个程序也许可行，但是考虑到以后实验六进程切换，最终也没有采取。最后还是在 c 模块声明结构体和指针，通过指针地址偏移的方式存储寄存器的值到相应位置，这是一个比较好的方式，因为结构体数组可以随意声明大小，为以后实验六的多进程切换也能做好铺垫。

第二个就是步骤四中关于 c 语言 io 库的设计，老师给的例子是 io 函数通过系统调用实现 io 操作，但是由于一些技术上的考虑（上文有讲述），我并没有按照这种形式来实现，而采用了 asm 内嵌汇编形式，实现的效果能达到完全相同

最后一个就是 c 库和 c 用户程序通过 ld 链接时会出现找不到入口的问题，我这里采用了一个简单的汇编模块实现入口功能，代码也没有几行。将该汇编模块与 c 库和 c 用户程序一起链接即可

而整个实验下来，也是存在着一些技术上的问题并没有得到完美地解决：

- 上文提到的 `restart()` 重载时会导致系统调用返回值（在寄存器 AX 处）被旧值覆盖，使得 `int 21h` 有返回值的系统调用一结束就被 `restart()` 覆盖，而无法把返回值传递给用户程序函数的问題，暂时无法解决（老师说实验七实现 `fork()` 实验会告诉你怎么办）

这里老师提到一个解决方法：把返回值写到 `save()` 的 AX 位置，但是显然不合理，会破坏原来 `save()` 保存的原有 AX 值，之后实验六的进程切换就会有错；同学提出一个解决方便：把返回值写入另一个内存位置，但是感觉难度偏大，要考虑不同的返回值类型，c 语言的 io 函数也不容易直接对内存进行读写操作，最后也没有考虑

- `printf("ch=%c, a=%d, str=%s", ch, a, str);` 这种类型的 io 函数实现难度过大，还是选择采用 `pchar()`，`pstr()`，`pint()` 分别实现单独输出字符、字符串、int 数值的功能