

实验五：实现系统调用

18340225 钟婕

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。

实验环境与工具

1、实验环境：

- 1) 实验运行环境：Windows10
- 2) 虚拟机软件：DosBox、VirtualBox

2、实验工具

- 1) 汇编编译器：NASM、TASM
- 2) C语言编译器：TCC
- 3) 文本编辑器：Visual Studio 2017
- 4) 软盘操作工具：WinHex
- 5) 调试工具：Bochs

实验方案与过程

• 实验思路与原理

总体思路

本次实验是在实验四的基础上，对于我们的整个原型操作系统做出进一步的扩展，包括对于中断处理的现场保护和现场恢复的Save和Restart过程的编写、对于仿照DOS用户程序返回方式的int 20h中断处理程序的编写，利用系统调用21h来实现自己封装的C语言的输入输出等库过程以及对于22h中断显示字符串处理程序的简单编写。

实验原理

根据实验内容和要求可知，本次实验是在实验四的基础上，增加中断处理的保护现场与恢复现场的Save和Restart过程，进一步理解中断机制，并且自己进行C语言的输入输出的库过程的设计，并编写21h的系统调用来实现对于输入、输出的过程的执行。根据课堂所学知识以及课后资料查询，将本人对于中断、系统调用的理解以及此次实验设计的知识、原理阐述如下：

1. 系统调用

操作系统除了执行用户的程序，还有义务为用户程序开发提供一些常用的服务。高级语言中，可以使用系统调用，实现软件重用的效果。操作系统提供的服务可以用多种方式供用户程序使用。

■ 子程序库静态链接：

采用子程序调用方式，如汇编语言中用call指令调用操作系统提供服务的子程序，静态链接到用户程序代码中，这种方式优点是程序执行快，最大缺点是用户程序内存和外存空间占用多，子程序库管理维护工作复杂。

■ 内核子程序软中断调用：

采用软中断方式，如汇编语言中用int指令调用操作系统提供服务的子程序，系统服务的子程序在内核，这种方式的优点是服务由系统提供，程序效率较高，且被所有用户程序代码共享，有利于节省内存，最大缺点是需要防止内核再入或内核设计为可再入，且用户程序陷入内核和内核返回用户程序的开销较大。

■ 子程序库动态链接：

采用动态链接技术，操作系统在运行时响应子程序调用，加载相应的子服务程序并链接致用户地址空间，这种方式优点是可由多方提供服务程序，服务更内容丰富，增加和变更服务方便，最大缺点是链接耗时多，程序响应变慢，实现复杂。

2. 中断处理程序

PC中断的处理过程包含：硬件实现的保护断点现场、执行中断处理程序以及返回到断点接着执行。

■ 硬件实现的保护断点现场

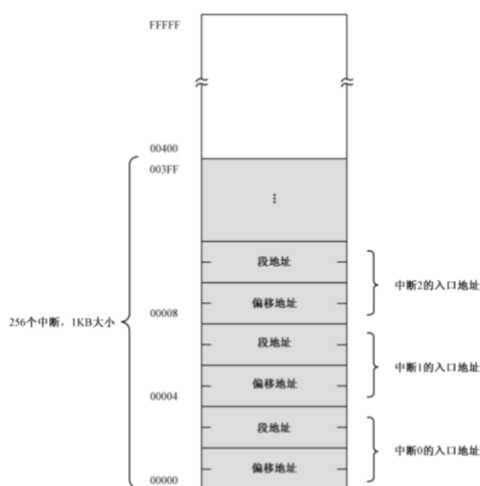
要将标志寄存器FLAGS压栈，然后清除它的IF位和TF位
再将当前的代码段寄存器CS和指令指针寄存器IP压栈
这是由硬件实现的过程，因此在编程中不涉及这方面的工作。

■ 执行中断处理程序

处理器根据已经拿到的中断号，它将该号码乘以4（毕竟每个中断在中断向量表中占4字节），就得到了该中断处理程序入口点在中断向量表中的偏移地址。

中断向量表：

X86计算机在启动的时候，会在内存的低位区（地址为0~1023，大小为1KB）创建含有256个中断向量的中断向量表，**中断向量其实就是中断处理程序的入口地址**，每个中断向量占4个字节，4个字节的具体分配为CS：IP（16位段地址：16位偏移地址），根据X86的小端存储方式，IP偏移地址存储在低字节，CS段地址存储在高字节。具体如下图：



处理器根据中断号在中断向量表中找到该中断号对应的中断处理程序入口点在中断向量表的偏移地址后，从表中依次取出中断程序的偏移地址和段地址，并分别传送到IP和CS。因此，处理器就开始执行CS：IP对应的入口点的中断处理程序了。

■ 返回到断点接着执行

所有中断处理程序的最后一条指令必须是中断返回指令iret。这将导致处理器依次从堆栈中弹出（恢复）IP、CS和FLAGS的原始内容，于是转到主程序接着执行。

至此就完成对于一次中断的响应，并返回到主程序继续执行。

3. 开中断与关中断

■ IF标志位

CPU是否响应在INTR(可屏蔽中断)线上出现的中断请求，取决于标志寄存器FLAGS中的IF标志位的状态值是否为1。可用机器指令STI/CLI置IF标志位为1/0来开/关中断。

■ STI/CLI指令

当我们进入中断处理程序执行的时候，此时仍然有可能有其他的中断请求，为了防止其他中断进入干扰程序的执行，我们可以选择屏蔽中断。使用指令CLI就可以修改处理器状态字中的中断相关的对应标志寄存器的值，实现中断屏蔽。如果我们结束了当前中断处理程序的执行时，需要开启中断，就可以使用STI指令。

另外，当我们编写的硬件中断处理程序结束的时候，我们需要给8259A芯片发送EOI信号来将存储当前正在处理的中断的ISR寄存器的对应位置清零，来结束中断服务程序。

4. 如何修改中断

根据上述描述，本人可将本次实验中断机制的实现归纳为：

- 1) 编写中断处理程序
- 2) 修改中断向量表

当然，对于键盘中断的修改仅限于执行用户程序之时，因此需要在读取扇区加载用户程序之前才修改中断向量表，在执行完用户程序返回时，需要还原中断向量表，否则会影响正常的输入命令操作！

5. 软中断实现系统服务

■ BIOS调用

其实它与内核子程序软中断调用方式原理是一样的

每一种服务由一个子程序实现，指定一个中断号对应这个服务，入口地址放在中断向量表中，中断号固定并且公布给用户，用户编程时才可以中断调用，参数传递可以使用栈、内在单元或寄存器。

■ 系统调用

1) 因为操作系统要提供的服务更多，服务子程序数量太多，但中断向量有限，因此，实际做法是专门指定一个中断号对应服务处理程序总入口，然后再将服务程序所有服务用功能号区分，并作为一个参数从用户中传递过来，服务程序再进行分支，进入相应的功能实现子程序。

2) 这种方案至少要求向用户公开一个中断号和参数表，即所谓的系统调用手册，供用户使用。

3) 如果用户所用的开发语言是汇编语言，可以直接使用软中断调用

4) 如果使用高级语言，则要用库过程封装调用的参数传递和软中断等指令汇编代码

我们的实验规定系统调用服务的中断号是21h。

• 编写中断处理的保护现场与恢复现场的Save和Restart过程

根据实验课以及理论课的所学知识可以知道，PC中断的处理过程包含：硬件实现的保护断点现场、执行中断处理程序以及返回到断点接着执行。在实验四时，我们简化了整个中断处理过程，我们只实现了中断处理程序的编写，在本次实验中我们需要考虑中断现场的保护与恢复，其实对于中断现场的保护与恢复主要涉及到对于被中断程序的所有上下文寄存器中的当前值的保护以及对于被中断程序的所有上下文寄存器中的在被中断前的值的恢复。通俗的理解就是，我们需要设计一个结构在中断处理程序开始前将被中断的程序的上下文寄存器的值都保护在结构中，接着执行中断处理程序，在执行完中断处理程序之后，我们需要将对应于被中断程序的结构中的所有上下文寄存器的值重新赋值给相应的寄存器，并利用iret指令返回到被中断程序的中断执行处

◦ PCB结构设计

由于需要保护被中断程序的所有上下文寄存器的值，包括：SS GS FS ES DS DI SI BP SP BX DX CX AX IP CS FLAGS。因此结合理论课学习的进程相关知识，设计一个寄存器的物理映像，包含上述所有寄存器作为数据成员，用以保存被中断程序的所有上下文寄存器的值。

结构设计如下所示：

```
//保存以下各个寄存器的值
typedef struct {
    int SS;
    int GS;
    int FS;
```

```

int ES;
int DS;
int DI;
int SI;
int BP;
int SP;
int BX;
int DX;
int CX;
int AX;
int IP;
int CS;
int FLAGS; //程序状态字PSW
}RegImg;

```

除此之外，考虑到在理论课学习的过程中，我们学习了进程的状态包含：就绪态、运行态和终止态这三个基本状态以及程序刚刚创建时的新建状态，因此除了上述的寄存器的物理映像，还在PCB进程控制块中加入了进程状态这一数据成员，用以保存当前被中断程序的状态。

因此整个PCB进程控制块的结构如下所示：

```

typedef struct
{
    RegImg RI; //寄存器物理映像
    int Process_Status; //进程状态
}PCB;

```

在编写好结构之后，我们需要充分利用结构来进行操作，由于我们的原型操作系统除却引导扇区程序外可以简要的分成内核程序与用户程序两大类，且内核程序 and 用户程序均有可能被中断，为了合理区分这两大类程序的PCB结构，我设计了一个数据类型为PCB的pcb数组，pcb[0]为内核程序的PCB，pcb[1]为用户程序的PCB，创建一个变量cur_pnum用以记录当前被中断的程序是内核程序还是用户程序，内核程序为0用户程序为1，同时还有NEW、Ready、Run、Exit等进程状态的值。

具体的数据设计如下：

```

int NEW = 0; //PCB结构刚刚初始化时进程状态是NEW
int Ready = 1; //就绪态
int Run = 2; //执行态
int Exit = 3; //退出态
PCB pcb[10]; //设计的数据类型为PCB的pcb数组，用以保存内核程序与用户程序的PCB
int cur_pnum = 0; /*0下标为内核程序，1为用户程序*/

```

○ PCB结构相关函数设计

在设计了上述的保存被中断程序的上下文寄存器的结构之后，需要设计相关的函数便于之后的Save和Restart过程的使用。

根据中断处理的保护现场和恢复现场的思想，我们很自然联想到需要设计将上下文寄存器值保存在被中断程序对于的PCB结构中的函数、与之对应的我们也需要设计将被中断程序的对的PCB结构返回给Restart的函数，再由Restart汇编重新赋值给各个寄存器。同时对于PCB结构的各个数据成员也需要进行初始化等操作。下面逐一介绍这些函数。

■ 初始化PCB数据成员函数

无论对于内核程序还是用户程序，我们均需要对于PCB的各个数据成员进行初始化，根据PCB的寄存器成员可知，对于段寄存器我们只需要利用程序的加载到内存的段地址进行初始化即可，而对于ip偏移量寄存器，由于我们的程序均是COM程序，因此初始时的偏移量为100h，而程序状态字则初始化为512，进程状态初始化为NEW，而对于其余的寄存器则初始化为0。

具体如下：

```
void initial(PCB* p,int segment,int offset)
{
    p->RI.GS = 0xb800; /*显存*/
    p->RI.SS = segment; //段寄存器初始化为段地址
    p->RI.ES = segment;
    p->RI.DS = segment;
    p->RI.CS = segment;
    p->RI.FS = segment;
    p->RI.IP = offset;
    p->RI.SP = offset - 4; //栈顶指针
    p->RI.AX = 0;
    p->RI.BX = 0;
    p->RI.CX = 0;
    p->RI.DX = 0;
    p->RI.DI = 0;
    p->RI.SI = 0;
    p->RI.BP = 0;
    p->RI.FLAGS = 512; //程序状态字初始化为512
    if (segment == 0x0800)
        p->Process_Status = Run;
    else
        p->Process_Status = NEW; //进程状态
}
```

特别注意的是，内核程序的进程状态不能初始化为NEW，因为内核程序在由引导扇区程序加载进入内存时即开始执行，而且在之后的Restart设计过程中，利用Bochs调试会发现，由于内核程序不是第一次运行，恢复内核程序的SP寄存器的值后由于上一次运行保存现场时还push了一些寄存器的值进入栈中，因此需要对恢复后的SP指针做一些修改，这是不同于用户程序的。具体问题与解决过程会在Restart部分说明。

■ 保存上下文寄存器值的函数

显然，对于Save过程中的保护被中断程序的所有上下文寄存器的值至PCB结构中，需要利用C语言实现，因此需要定义一个Save_PCB函数用以将所有上下文寄存器的值。特别的由于被中断程序的ip、cs、psw是由调用中断时自动入栈的，但由于call Save()过程时，会将返回的偏移量也入栈，因此其余寄存器的保存与ip、cs和psw三个数据的保存需要区分开(因为栈中还有Save过程的返回偏移地址)。因此分别设计了Save_PCB、Save_PSP两个函数分别保存。

具体如下：

```
void Save_PCB(int gs, int fs, int es, int ds, int di, int si, int bp,
              int sp, int dx, int cx, int bx, int ax, int ss)
{
    pcb[cur_pnum].RI.AX = ax; //cur_pnum为当前被中断程序在pcb数组的下标
    pcb[cur_pnum].RI.BX = bx; //依次将各个寄存器的值保存
    pcb[cur_pnum].RI.CX = cx;
    pcb[cur_pnum].RI.DX = dx;
```

```

        pcb[cur_pnum].RI.DS = ds;
        pcb[cur_pnum].RI.ES = es;
        pcb[cur_pnum].RI.FS = fs;
        pcb[cur_pnum].RI.GS = gs;
        pcb[cur_pnum].RI.SS = ss;

        pcb[cur_pnum].RI.DI = di;
        pcb[cur_pnum].RI.SI = si;
        pcb[cur_pnum].RI.SP = sp;
        pcb[cur_pnum].RI.BP = bp;
    }

    void Save_PSP(int ip, int cs, int flags)//保存ip\cs\psw的值
    {
        pcb[cur_pnum].RI.IP = ip;
        pcb[cur_pnum].RI.CS = cs;
        pcb[cur_pnum].RI.FLAGS = flags;
    }

```

■ 切换内核程序与用户程序的函数

由于内核程序中的int 22h的中断执行时需要Save和Restart过程，之后就是int 20h用户程序返回时需要Save和Restart过程，由于用户程序是由键入的命令指定执行的，执行次数和顺序不定，而内核的22h的中断是由内核程序指定调用的，因此在内核程序调用完int 22h执行完之后，也即对于pcb[0]对应于内核程序的中断执行完之后，需要将cur_pnum加一，准备为之后的键入的执行用户程序的命令的Save和Restart过程做准备。

具体代码如下：

```

void Exchange()
{
    if (cur_pnum == 0)//将当前内核程序的PCB块切换成用户程序的PCB块
        cur_pnum = 1;
}

```

■ 获取当前被中断程序对应PCB结构的函数

由于在Restart过程中，需要利用被中断程序的PCB结构中的值恢复现场，因此需要设计一个函数返回在pcb数组中对应于当前被中断程序的PCB结构。函数设计相对简单，只需要根据cur_pnum(即当前被中断程序对应的PCB结构在pcb数组的下标)返回对应pcb数组元素的地址即可。

具体代码如下：

```

PCB* Current_PCB()
{
    return &pcb[cur_pnum]; //根据cur_pnum的值返回地址
}

```

○ Save过程设计

根据上述的数据结构设计以及相关的函数设计，对于Save过程的操作其实就可以简化成调用上述的Save_PCB和Save_PSP过程即可。当然由于Save过程是在内核程序的汇编实现的底层模块中编写的，因此需要考虑汇编模块调用C模块的参数入栈与出栈的过程，根据实验三的知识以及实践经验可知，汇编模块调用C模块时只需要要遵从将参数按照C函数声明的右参先入栈的顺序即可。

另外，由于调用中断时，会将被中断程序中中断的ip、cs、PSW均入栈，而且在中断处理程序执行前，调用Save过程是使用call指令的，因此Save过程的返回偏移地址也会入栈，此时栈中为PSW\CS\IP\SaveIP因此如果想要直接利用一个函数将被中断程序的所有上下文寄存器入栈是行不通的，因此根据上述，设计了Save_PCB和Save_PSP两个过程分别用以保存除ip、cs、PSW以外的寄存器的值以及ip、cs、PSW的值。

首先需要先按照右参先入栈的顺序，将Save_PCB函数声明的参数按从右到左的顺序对应的寄存器的值依次入栈，然后利用call指令调用Save_PCB函数，特别注意地是，需要养成良好的习惯，再调用完Save_PCB函数之后需要将参数依次逆序出栈，这样栈中才能恢复到调用之前的PSW\CS\IP\SaveIP状态。

部分代码如下所示：

```
Save:
    push ss;按照Save_PCB函数的声明参数从右到左的顺序依次将参数入栈
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs;要记得重新给段寄存器赋值！！否则调用错误！
    mov es,ax
    mov ds,ax
    ;汇编调用c模块参数传递之后要自己出栈！！！
    call near ptr _Save_PCB;中断调用是模式切换的时机，因此要将被中断的进程的上下文保存在该进程的进程控制块中
    ;逆序依次将参数出栈！！！
    .386
    pop gs
    pop fs
    .8086
    pop es
    pop ds
    pop di
    pop si
    pop bp
    pop sp
    pop dx
    pop cx
    pop bx
    pop ax
    pop ss
```


特别注意地是，由于涉及到段内过程调用，因此在call指令调用函数时，一定要记得重新给段寄存器赋值，以防万一。在用Bochs单步调试测试过程中，最初发现用户程序进入20h中断时会出现一直无法成功调用Save_PCB函数的情况，后来经过Bochs单步调试以及利用sreg命令查看cpu寄存器的值之后，才知道原来是由于用户程序的段地址与内核程序的段地址是不同的，当用户程序返回时，CS寄存器的值会更新成内核的段地址，但此时ES、DS寄存器的值还是保留着原来用户程序的段地址，因此会出现错误。因此在调用之前需要利用已更新的CS寄存器的值重新给ES、DS寄存器赋值，这样才能确保调用过程的顺利执行。

接着，在保存完上述除ip\cs\psw的上下文寄存器的值以后，栈中状态为PSW\CS\IP\SaveIP，因此此时需要先将Save过程的返回地址出栈，保存在一个中间变量Back中，接着再调用Save_PSP就能够保存PSW\CS\IP的值至被中断的程序的对应的PCB结构中。在保存完PSW\CS\IP这三个值之后，我们需要将刚刚保存在中间变量的Back中的Save函数的返回地址重新入栈，以便于Save函数的ret指令取出返回的偏移地址，回到中断处理程序。

下面展示调用Save_PSP并返回中断处理程序的代码：

```
mov ax,cs;要记得重新给段寄存器赋值！！否则调用错误！
mov es,ax
mov ds,ax
pop ax;弹出栈中的ip2即Save函数的返回地址,此时栈中psw/ip1/cs1
;不能用ax存ax在函数中会变化！！！
;用一个中间变量保存栈中的Save返回地址
push bp
mov bp,offset Back
mov word ptr [bp],ax
pop bp

call near ptr _Save_PSP;调用函数将psw、cs、ip保存在被中断程序的结构中

;栈中的psw\cs\ip可以不用出栈
;需要将刚刚保存在Back中的返回地址重新入栈，便于返回
push bp
mov bp,offset Back
mov ax,word ptr [bp]
pop bp

push ax;再把ip2入栈，此时栈中ip2
ret;返回
```

根据上述描述就可以成功将被中断程序的所有上下文寄存器的值保存在对应的PCB结构中。

下面用Bochs调试验证：已经将被中断程序的对应的寄存器的值均保存在被中断程序对应的结构中。

以下是被中断程序刚刚被中断时的所有寄存器的值：

```
Bochs for Windows - Console
00008257: ( ) : pop ss ; 17
<bochs:14> reg
rax: 0x00000000_00000000 rcx: 0x00000000_00090002
rdx: 0x00000000_00000000 rbx: 0x00000000_00000000
rsp: 0x00000000_0000fff3 rbp: 0x00000000_00000000
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00000231
eflags 0x00000082: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:15> sreg
es:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0800, dh=0x00009300, dl=0x8000ffff, valid=7
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0800, dh=0x00009300, dl=0x8000ffff, valid=7
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x000000000000f1b7, limit=0x30
idtr:base=0x0000000000000000, limit=0x3ff
<bochs:16>
```

调用Save_PCB\Save_PSW时的栈中内容以及调用之后查看数据结构PCB的内容:

```
Bochs for Windows - Console
00008255: ( ) : pop bx ; 5b
00008256: ( ) : pop ax ; 58
00008257: ( ) : pop ss ; 17
<bochs:14> lb 0x00008246
<bochs:15> c
(0) Breakpoint 3, 0x0000000000008246 in ?? ()
Next at t=158786832
(0) [0x000000008246] 0800:0246 (unk. ctxt): call .+520 (0x00008451) ; e80802
<bochs:16> print-stack
Stack address size 2
STACK 0x17fd9 [0x0000]
STACK 0x17fdb [0x0000]
STACK 0x17fdd [0x0800]
STACK 0x17fdf [0x0800]
STACK 0x17fe1 [0xffac]
STACK 0x17fe3 [0x0000]
STACK 0x17fe5 [0x0000]
STACK 0x17fe7 [0xffe9]
STACK 0x17fe9 [0x0000]
STACK 0x17feb [0x0002]
STACK 0x17fed [0x0000]
STACK 0x17fef [0x0000]
STACK 0x17ff1 [0x0800]
STACK 0x17ff3 [0x0378]
STACK 0x17ff5 [0x022c]
STACK 0x17ff7 [0x0800]
<bochs:17>
```

```
Bochs for Windows - Console
000082d0: ( ) : pop cx ; 59
000082d1: ( ) : pop dx ; 5a
000082d2: ( ) : pop bx ; 5b
000082d3: ( ) : pop bp ; 5d
<bochs:43> lb 0x00008280
<bochs:44> c
(0) Breakpoint 10, 0x0000000000008280 in ?? ()
Next at t=208791045
(0) [0x000000008280] 0800:0280 (unk. ctxt): mov bp, ax ; 8be8
<bochs:45> s
Next at t=208791046
(0) [0x000000008282] 0800:0282 (unk. ctxt): mov ss, word ptr ds:[bp] ; 3e8e5600
<bochs:46> reg
rax: 0x00000000_00000cf0 rcx: 0x00000000_00090002
rdx: 0x00000000_00000000 rbx: 0x00000000_00000000
rsp: 0x00000000_0000fff5 rbp: 0x00000000_00000cf0
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00000282
eflags 0x00000006: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af PF cf
<bochs:47> xp /16wx 0x0800:0xcf0
[bochs]:
0x0000000000008cf0 <bogus+ 0>: 0x00000800 0x08000000 0xffac0800 0x00000000
0x0000000000008d00 <bogus+ 16>: 0x0000ffe9 0x00020000 0x022c0000 0x00820800
0x0000000000008d10 <bogus+ 32>: 0x12000002 0x1200b800 0x12001200 0x00000000
0x0000000000008d20 <bogus+ 48>: 0x00fc0000 0x00000000 0x00000000 0x12000100
<bochs:48>
```

调用Save_PCB、Save_PSW函数后
查看PCB数据结构内容

Restart过程设计

根据上述Save过程设计的思想, 那么对于Restart而言其实就是将在Save过程中保存在被中断程序对应PCB结构的各个寄存器的值重新赋值给各个寄存器即可。特别注意的是对于ip、cs和psw这三个值则不需要特别的进行赋值操作, 我们也不能对cs、ip和psw直接赋值, 我们只需要将这三个保存在被中断程序中的值重新入栈待Restart过程的最后的iret指令执行跳转回被中断程序的中断处即可。

值得注意的是，iret指令是Restart过程中最后执行的指令，根据栈先进后出的特点，因此需要将中断程序的PCB结构中的ip、cs、psw先依次入栈，接着才是其他寄存器。

那么如何获得被中断程序的PCB结构呢？根据在PCB的相关函数设计的描述可知，我们可以利用Current_PCB函数的返回值获取当前被中断程序的PCB结构。由C语言与汇编语言混合编程的原理可知，当汇编模块调用C模块的函数时，若C函数存在返回值，则返回值默认保存在ax寄存器中，因此我们只需要利用call指令调用Current_PCB函数并将ax寄存器的值赋值给bp寄存器，此时bp寄存器的值是当前被中断程序的PCB结构在内存中的起始偏移地址，因此我们利用ds数据段寄存器作为段地址，采用段地址：偏移地址的方式，逐个访问即可。特别是，我们需要按照PCB结构中定义的顺序计算偏移量，因为每个数据成员均是int类型的，偏移量计算以字节为单位，所以从第一个数据成员偏移地址即为起始偏移地址开始，每访问下一个数据成员偏移地址就要加2,以此类推。

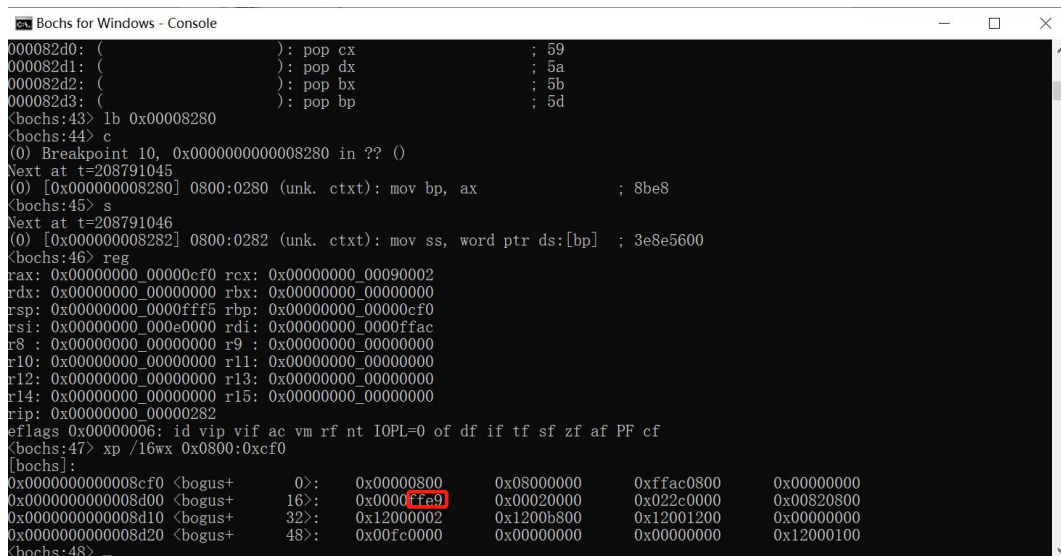
在这里有一个易忽略的问题，对于栈的访问是跟ss寄存器以及sp栈顶指针有关的，在Restart过程中我们需要借助栈来恢复各个寄存器的值，但我们不能破坏中断处理程序中的栈的内容，需要养成良好习惯，先将PCB结构中对应的ss、sp寄存器的值取出并给ss、sp寄存器重新赋值，切换成当前需要重启的程序的栈中。因此需要先利用mov指令完成对ss、sp的恢复。

除此之外，在刚刚编写程序并利用Bochs进行调试的时候，发现：直接将当前要重启的程序对应的PCB结构中的sp的值赋值给sp寄存器，并对栈进行访问，会出现栈中内容增加了一些数据的问题，后来发现如果当前要重启的程序是第一次运行，则sp的值是正确的，但如果当前要重启的程序不是第一次运行，即之前被中断过，则sp的值需要加上18才是正确的值。

这是为什么呢？我刚开始百思不得其解，后来通过反复的回看代码以及用Bochs调试比对，我发现，如果不是第一次运行的程序，则说明被中断过，那么就一定经过了Save过程，通过Save过程我们可以发现，我们本来应该保存的是被中断程序一中断时的所有上下文寄存器的值，但在调用Save_PCB函数之前push参数的时候，我们在push sp寄存器的值之前还push了ss寄存器的值，ax~dx寄存器的值，而且在push 这些寄存器的值之前，由于中断调用ip\cs\psw的值也已入栈，且call指令调用Save过程时Save返回的偏移地址也入栈，因此在push sp寄存器的值之前，我们还push了9个字大小的值入栈，因此当我们push栈指针sp的时候其实sp的值已经跟真正的sp的值相差了18，因此要想获得正确的sp寄存器的值，我们需要将PCB结构中对应的保存的值加上18才是真正的值。

下面展示Bochs调试发现问题的过程：

用以保存的数据结构PCB中的sp的值为图中红框：



```
Bochs for Windows - Console
000082d0: ( ) : pop cx ; 59
000082d1: ( ) : pop dx ; 5a
000082d2: ( ) : pop bx ; 5b
000082d3: ( ) : pop bp ; 5d
<bochs:43> lb 0x00008280
<bochs:44> c
(0) Breakpoint 10, 0x0000000000008280 in ?? ()
Next at t=208791045
(0) [0x0000000000008280] 0800:0280 (unk. ctxt): mov bp, ax ; 8be8
<bochs:45> s
Next at t=208791046
(0) [0x0000000000008282] 0800:0282 (unk. ctxt): mov ss, word ptr ds:[bp] ; 3e8e5600
<bochs:46> reg
rax: 0x00000000_00000cf0 rcx: 0x00000000_00090002
rdx: 0x00000000_00000000 rbx: 0x00000000_00000000
rsp: 0x00000000_0000fff5 rbp: 0x00000000_00000cf0
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00000282
eflags 0x00000006: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af PF cf
<bochs:47> xp /16wx 0x0800:0xc0
[bochs]:
0x00000000000008cf0 <bogus+ 0>: 0x00000800 0x08000000 0xffac0800 0x00000000
0x00000000000008d00 <bogus+ 16>: 0x0000ffa9 0x00020000 0x022c0000 0x00820800
0x00000000000008d10 <bogus+ 32>: 0x12000002 0x1200b800 0x12001200 0x00000000
0x00000000000008d20 <bogus+ 48>: 0x00fc0000 0x00000000 0x00000000 0x12000100
<bochs:48>
```

正确的SP的值应该为如下图刚刚被中断时的sp值：

```
Bochs for Windows - Console
0000892c: (          ): mov ax, 0x0038          ; b83800
0000892f: (          ): push ax                     ; 50
<bochs:8> n
Next at t=158786813
(0) [0x000000008926] 0800:0926 (unk. ctxt): call .-1791 (0x0000822a) ; e801f9
<bochs:9> s
Next at t=158786814
(0) [0x00000000822a] 0800:022a (unk. ctxt): int 0x22                ; cd22
<bochs:10> s
Next at t=158786815
(0) [0x000000008375] 0800:0375 (unk. ctxt): call .-327 (0x00008231) ; e8b9fe
<bochs:11> print-stack
Stack address size 2
STACK 0x17ff5 [0x022c]
STACK 0x17ff7 [0x0800]
STACK 0x17ff9 [0x0082]
STACK 0x17ffb [0x0929]
STACK 0x17ffd [0x0136]
STACK 0x17fff [0x0000]
STACK 0x18001 [0x0000]
STACK 0x18003 [0x0000]
STACK 0x18005 [0x0000]
STACK 0x18007 [0x0000]
STACK 0x18009 [0x0000]
STACK 0x1800b [0x0000]
STACK 0x1800d [0x0000]
STACK 0x1800f [0x0000]
STACK 0x18011 [0x0000]
STACK 0x18013 [0x0000]
<bochs:12>
```

通过比较：17ff5与17fe9相差18，因此发现问题。

那么如何区分当前程序是否是第一次运行呢？我很自然的联想到之前在PCB结构设计时的进程状态变量，由于内核程序一旦由引导扇区程序加载后便开始执行，因此内核程序不是第一次运行程序状态为Run，而对于用户程序在没有键入命令调用时是为运行的，因此初始化为NEW。因此我们可以在重启时通过比较PCB结构中的进程状态数据是否为NEW来判断程序是否是第一次运行，若是则直接将PCB结构的sp值赋值给SP寄存器即可，若不是则需要将sp的值加上18。

具体代码如下：

Restart:

```
mov ax,cs;要记得重新给段寄存器赋值！！否则调用错误！
mov es,ax
mov ds,ax
;获取要重启的程序的PCB结构地址
call near ptr _Current_PCB;返回值保存在ax中
mov bp,ax
```

;要先恢复ss\sp的值，不然会出现栈错误！！！！

```
mov ss,word ptr ds:[bp+0]
mov sp,word ptr ds:[bp+16]
```

```
cmp word ptr ds:[bp+32],0 ;查看当前状态是不是new
jnz Not_First_Time ;如果是new状态说明是第一次运行，否则需要更改sp的值
```

Not_First_Time:

```
add sp,18 ;如果不是第一次运行，此时从PCB中得到的sp不一定是正确的值，为了保险起见取第一次的值
jmp redone
```

接下来，就可以按照上述关于恢复寄存器值的分析，借助栈完成恢复现场，依次将psw、cs、ip以及除psw、cs、ip、ss、sp寄存器的值入栈（psw、cs、ip的偏移地址分别为：30、28、26），再按照入栈的逆序除psw、cs、ip外出栈赋值给各个寄存器即可。

出栈完成之后，最后用一条iret指令返回要重启的程序被中断时的地址继续执行即可。

具体代码如下：

redone:

```

call near ptr _Exchange
; 没有push ss 和 sp的值因为已经赋值了
;取出PCB中的值, 恢复现场
;flags,cs,ip依次入栈, iret时自动取出
push word ptr ds:[bp+30]
push word ptr ds:[bp+28]
push word ptr ds:[bp+26]

push word ptr ds:[bp+2]
push word ptr ds:[bp+4]
push word ptr ds:[bp+6]
push word ptr ds:[bp+8]
push word ptr ds:[bp+10]
push word ptr ds:[bp+12]
push word ptr ds:[bp+14]
push word ptr ds:[bp+18]
push word ptr ds:[bp+20]
push word ptr ds:[bp+22]
push word ptr ds:[bp+24]

pop ax
pop cx
pop dx
pop bx
pop bp
pop si
pop di
pop ds
pop es
.386
pop fs
pop gs
.8086

iret

```

• INT 20H：用户程序返回内核程序中断处理程序的编写

根据本次实验的要求，需要利用20h中断实现COM格式用户程序返回内核程序的功能，即类似于DOS的做法在COM格式的用户程序偏移量为0处预设int 20h的指令，在用户程序运行结束时利用ret指令段内转移至偏移量为0处调用20h中断，而20h中断的处理程序即是用户程序返回内核程序的那一段代码，因此我们可以巧妙地利用中断顺利地返回内核程序。

设计思路即为：把原来的加载用户程序并跳转执行后返回的那部分代码作为int 20h的中断处理程序即可，同时需要在用户程序偏移量为0处预设一条int 20h的指令，在用户程序结尾处先push偏移量0，再执行ret指令转移到偏移量为0处执行int 20h指令即可。不要忘记修改中断向量表，在加载用户程序并跳转执行的模块开始时，根据实验四的修改中断向量方法修改20h的中断向量。

下面展示修改中断向量部分以及int 20h的中断处理程序：

```

xor ax,ax
mov es,ax
mov word ptr es:[32*4],offset int20h1
mov word ptr es:[32*4+2],cs

```



```

int20h1:
    call near ptr Save;调用上述Save过程保护现场
    mov ax,cs;一定要记得切换段寄存器的值，因为COM格式用户程序的段地址与内核不同
    mov ds,ax
    mov es,ax
    mov ss,ax;加载用户程序执行的过程的返回地址在系统栈中，因此要从用户栈切换回系统栈
    pop bp
    pop es
    pop ds ;恢复现场，逆序出栈
    pop ax
    ret;加载用户程序并执行模块的返回
    call near ptr Restart;调用Restart过程恢复现场，其实在此无用

```

接下来，展示用户程序预设int 20h指令并用ret指令跳转至偏移量为0处执行int 20h的代码。int 20h指令的机器码为CDH 20H，每条指令占两个字节，低地址存放指令码，在es:0处存放这条指令即可。在预设完指令之后，在用户程序结束处需要先将偏移量0入栈，这样ret指令才能近转移至段内偏移量0处。具体如下：

```

mov ah,20H;预设int 20h指令
mov al,0CDH
mov word[es:0],ax
mov ax,0
push ax
ret;ret指令用栈中的数据，修改IP的内容，从而实现近转移

```

• C语言输入输出库设计libs.asm

根据实验要求，我们需要创建一个libs.asm文件用于C语言输入输出库的设计，主要需要封装的库函数有：getch、putch、gets、puts、scanf、printf。根据前面几次实验的过程我们很自然的联想到输入、输出一个字符的函数的实现可以充分利用BIOS的功能调用来实现，而对于输入、输出一个字符串的函数设计，则可以通过调用输入、输出一个字符的函数以及充分利用C语言与汇编语言混合编程模块调用的特点来实现！

同时，根据实验要求，将上述的getch、putch、gets、puts、scanf、printf六个模块利用系统调用来实现调用，系统调用int 21h的具体实现方法会在下一部分具体介绍。

下面将对于这六个C语言输入、输出库函数分别做出介绍：

◦ 输入一个字符getch

我们设定getch的函数有一个参数&Ch,, 即代表着传入的为保存输入的字符的变量Ch的地址。

根据BIOS功能调用的功能表可知，我们可以利用BIOS的中断号为16h号，功能号为0的功能调用实现输入一个字符，输入的字符保存在AL中。接着，我们充分利用实验三学习的C语言与汇编语言混合编程模块调用的参数传递方法，利用bp寄存器访问getch函数的参数，并将保存在al寄存器中的输入的字符保存在参数指定的内存地址中即可。

具体代码如下所示：

```

getch:
    mov ax,cs;置其它寄存器与CS相同
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = CS
    push bp;将一些会被改变的寄存器的值入栈，做保护

```

```

push bx
mov bp, sp
mov bx, [bp+12];char/ip1/ip2/cs/psw/bp/bx, getch是实现在系统调用int 21h
下的, 因此栈中保存在系统调用入栈的psw、cs、ip
;利用中断号16h的0号功能实现输入一个字符
;输入的字符保存在al寄存器中
mov ah, 0 ;功能号
int 16h
;bx为传入的保存输入字符变量的偏移地址
mov byte ptr [bx], al ;用一个字符指针保存输入的字符
mov sp, bp
pop bx
pop bp
jmp near ptr Restart
iret

```

○ 输出一个字符putch

由于在实验三时, 本人已经封装相关输入输出函数, 因此积累了些许经验。实现输出一个字符的过程, 通过在BIOS功能调用中查找可以发现存在一个中断号为10h功能号为0EH的输出一个字符的功能调用, 因此在设计中直接利用这个功能调用即可。

输出一个字符的过程, 有一个字符类型的参数, 由调用者通过栈传递, 由于过程调用时会将会IP压栈, 且由于系统调用也会把中断的程序状态字、程序偏移地址以及程序段地址压栈, 因此访问参数不仅不能直接利用栈指针SP, 需要利用BP, 而且需要找对参数位置。同时, 出于保护BP寄存器的目的, 也需要将BP寄存器的原值入栈, 因此利用BP寄存器访问栈中参数时, 需要从[BP+10]处访问。

下面展示具体代码:

```

putch:
mov ax, cs;置其它寄存器与CS相同
mov ds, ax; DS = CS
mov es, ax; ES = CS
mov ss, ax; SS = CS
push bp ;保护bp的值, 因为接下来要用bp去访问栈(参数在栈中)
mov bp, sp ;栈顶指针sp赋给bp
mov ax, [bp+10];char/ip1/ip2/cs/psw/bp
mov ah, 0EH ;ah为功能号
mov bl, 0h ;前景色(图形模式)
int 10h
mov sp, bp
pop bp ;恢复bp的值
jmp near ptr Restart
iret ;返回

```

○ 字符串输入gets

根据实验要求, 设计了一个gets函数, 设计函数有一个参数即用以保存输入的字符串的一个字符串指针, 由于要想直接利用纯汇编进行字符串的输入是很困难的, 因此我充分利用C语言与汇编语言混合编程的优势。除此之外, 由于字符串输入的直接实现是比较复杂的, 因此联想到字符串其实就是由一个个字符组成的, 由此我们可以用C语言编写, 利用之前实现的getch函数一个个读入字符组成字符串, 再保存在变量中即可。

因此对于汇编过程gets的实现, 是利用call指令调用C模块中的getstr函数, 将gets的参数即字符串指针也利用push语句传入, 作为getstr函数的参数, 在getstr中实现按一个个字符输入, 即利用下标一次次调用getch函数输入字符, 保存在对应下标的字符串数组位置。

在程序测试运行的时候发现，由于是字符单个单个输入，很容易出现输入错误的情况，通过网上查询资料发现BIOS输入backspace退格字符时候，只会将光标后移。因此我添加了退格的操作，通过判断输入字符是否为退格，若为退格，如果不是输入的第一个字符，那么就先输出退格，让光标前移，再调用汇编过程输出一个空格，光标后移，接着再输出一个退格让光标前移，重新调用汇编输入一个字符的过程，再次输入一个字符。记得将存储字符的数组下标前移一个单位，用新输入的字符覆盖原来的字符。而如果是输入的第一个字符就为退格那么重新输入即可。字符串的输入以回车结束，记得最后将字符串末尾加上'\0'。

除此之外，为了输入的可视化，还在每次输入一个字符调用完getch函数之后调用putch函数将刚刚输入的字符显示出来，增强界面可观性。

下面展示getstr函数的代码：

```
void getstr(char * s)
{
    int index = 0;
    getch(&Ch);
    while (Ch != 13)
    {
        if (Ch == 8)
        {
            if (index != 0)
            {
                putch(Ch);
                putch(32);
                index--;
                putch(Ch);
                getch(&Ch);
                continue;
            }
            else
            {
                getch(&Ch);
                continue;
            }
        }
        putch(Ch);
        s[index] = Ch;
        index++;
        getch(&Ch);
    }
    s[index] = '\0';
}
```

接下来显示gets汇编过程如何调用C模块函数getstr，首先将段寄存器赋值确保成功调用C模块函数，然后利用bp访问栈中的参数，再将栈中的蚕食重新入栈，作为getstr函数的参数。


```

gets:
    mov ax,cs;置其它寄存器与CS相同
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = CS
    push bp
    mov bp,sp
    mov bx,[bp+10]
    push [bp+10]
    call near ptr _getstr
    pop bp
    pop bp
    iret

```

○ 字符串输出puts

类似于gets函数的设计，要想直接利用纯汇编进行字符串的输入是很困难的，因此我充分利用C语言与汇编语言混合编程的优势。除此之外，由于字符串输出的直接实现是比较复杂的，因此联想到字符串其实就是由一个个字符组成的，由此我们可以用C语言编写，利用之前实现的putch函数一个个输出字符即可。

因此需要利用putch过程，除此之外由于对于字符串按下标的访问，对于C语言来说相对简单，因此同样类似于gets的思想，在C模块调用putch函数实现一个个字符的输出函数putstr，基本思想为：参数传入一个字符数组，通过一个while循环判断当前下标对应的字符是否为'\0'，如果不是，则调用汇编输出一个字符的过程，输出这个字符；反之退出循环。

而在汇编的puts过程则将栈中传入的待输出的字符串指针重新入栈，然后call指令调用C模块putstr函数即可。

下面展示putstr函数代码：

```

void putstr(char *str)
{
    int i = 0;
    while (str[i] != 0)//判断结束标志
    {
        putch(str[i]); //输出字符
        i++;
    }
}

```

puts过程代码：

```

puts:
    mov ax,cs;置其它寄存器与CS相同
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = CS
    push bp
    mov bp,sp
    mov bx,[bp+10];保存要显示的字符串的起始地址
    push [bp+10]
    call near ptr _putstr
    pop bp
    pop bp
    iret

```

o 字符串输出printf

其实字符串输出函数printf的功能与字符串输出函数puts的功能相同，因此对于printf函数的实现可以参考puts函数的实现，其实也就是同样地利用putch过程一个一个地输出字符从而达到输出字符串的效果，而逐次一个个地输出字符通过C语言进行功能实现比较简单，因此同样利用C语言来实现，具体代码如下所示：

```
void printf(char *str)
{
    int i = 0;
    while (str[i] != 0)
    {
        putch(str[i]);
        i++;
    }
}
```

o 字符串输入scanf

而对于字符串输入函数scanf，其实它就是通过类型符，实现了包含getch以及gets的功能，由于前面已经实现了这两个库过程，因此只需要在此通过输入的类型符判断是输入一个字符还是输入一个字符串即可，scanf函数有两个参数，包含：第一个参数说明输入类型的字符串以及第二个参数存储输入的内容的变量。通过bp指针，可以获取这两个参数，再用C语言设计函数scanfstr进行输入类型判断，即根据第一个参数（类型说明符）判断是调用getch函数还是gets函数，因此只需要栈中的对应位置的参数入栈作为scanfstr的参数即可，就可以充分利用getch函数和gets函数进行输入。

下面展示C语言编写的区分输入字符与输入字符串的具体代码：

```
void scanfstr(char*s1, char*s2)
{
    if (s1[0] == '%'&&s1[1] == 'd')
        getch(s2);
    if (s1[0] == '%'&&s1[1] == 's')
        gets(s2);
}
```

根据上述，就可以完成scanfstr的函数设计，接下来在汇编中实现scanf的对于scanfstr的调用。在本次实验设计中，我将scanf的第一个参数只是作为类型说明符判断是字符输入还是字符串输入而已，而对于第二个参数的含义则跟前面所述的gets、getch的含义类似，是用来保存输入数据的变量。

下面展示scanf的汇编模块：如何获取参数含类型符与保存数据的变量。

```
scanf:
; %的ASCII码值为37
mov ax, cs; 置其它寄存器与CS相同
mov ds, ax; DS = CS
mov es, ax; ES = CS
mov ss, ax; SS = CS
push bp
mov bp, sp
mov bx, [bp+10]; char/str/ip1/ip2/cs/psw/bp 获取类型说明符
mov cx, [bp+12]; c模块调用汇编右参先入栈! 获取保存输入数据的变量
```

```
push [bp+12];将这两个参数先后入栈，准备调用上述的C模块scanfstr函数
push [bp+10]
call near ptr _scanfstr;调用scanfstr函数判断是输入字符还是字符串，充分利用
getch、gets
pop bp;参数出栈
pop bp
pop bp
iret
```

对于上述的六种C库的输入输出函数的设计的有一个关键也是一个坑点，即根据实验要求，对于这六种函数的调用需要基于int 21h的系统调用设计（关于如何设计系统调用，将在接下来的部分详述），因此根据《实验原理》部分的中断调用过程原理可知，中断触发时，系统会将被中断程序的程序状态字PSW、中断点的偏移地址IP以及中断程序的段地址CS依次入栈，因此在利用bp指针从栈中获取参数时需要考虑栈中还有上述三个数据，因此需要更改访问的偏移量才能访问到正确的数据，获取想要的参数。

在刚开始设计的时候，我忽略了这一点，因此老是出错，再利用Bochs调试，利用print-stack命令查看栈内容的时候发现栈中内容多了好几个，因此才意识到原来自己忽略了中断调用时的相关数据入栈。这点非常关键。

• INT 21H系统调用

根据实验原理部分的叙述可知，因为操作系统要提供的服务很多，服务子程序的数量太多，而中断向量的数量有限，因此我们应该专门指定一个服务处理程序的总入口，然后再将所有的服务程序按照功能号进行区分，并作为一个参数从用户中传递过来，根据功能号进入相应的服务子程序。

在本次实验中，将C模块的库的输入、输出函数的程序作为系统调用的多个服务子程序，统一用int 21h作为中断号，再利用ah寄存器进行功能号的参数传递，对于多个输入、输出函数分配唯一的功能力号，根据功能号选择对应的服务子程序执行。

下面分成两部分描述int 21h的系统调用：

◦ 21h号中断服务程序

根据系统调用的功能描述，我们需要将自己的封装编写的C库的输入、输出函数程序统一在21h号的中断服务程序中，再根据ah中描述的功能号的值进行不同子程序的选择。在之前所述的C库输入、输出函数封装中，我们实现了getch、putch、gets、puts、printf以及scanf六个输入输出函数，同时加上早在实验三就已经实现的清屏cls函数，总共有六个子程序，因此我们依次设定功能号：

- 0——cls；清屏操作
- 1——getch；输入字符
- 2——putch；输出字符
- 3——puts；输出字符串
- 4——gets；输入字符串
- 5——scanf；输入字符或者字符串

我们将上述的六个函数主体分别组织成六个程序段，每个程序段即对应一个子程序功能，在21h号的中断服务程序入口处，利用cmp指令、jnz指令与jmp指令结合通过对ah寄存器中保存的功能号的判断，选择跳转到不同的子程序段，由于是中断调用程序，因此每个程序段执行结束返回时，需要利用iret指令返回而不能用ret指令。

而对于原来的各个函数的汇编过程声明，则只需要在每个过程中对ah寄存器赋予相应的功能号，并使用int指令调用21h号中断，进入21h号的中断服务程序，在中断服务程序中根据ah寄存器中的值，判断各个功能号对应的子程序，进入对应的子程序执行并用iret指令返回即可。

在每个声明为public的过程段中，只需要先按照上述的功能号表格中的对应关系，对ah进行相应的功能号赋值并且利用int指令中断调用21h号中断即可。

下面以getch过程为例，展示过程调用21h中断的代码：

```
;调用中断号16h的01h功能读入一个字符
public _getch
_getch proc
    mov ax,cs;置其它寄存器与CS相同
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = CS
    mov ah,1;功能号赋值
    int 21h;调用21h中断
    ret
_getch endp
```

下面展示中断服务程序入口的功能号匹配代码：

```
int21h:
    call near ptr Save;保护现场
    cmp ah,0;将ah中的值依次与0~5作比较
    jnz Ch1;若匹配则跳转至相应的程序段
    jmp cIs;若不匹配则进行下一次的比较

Ch1:
    cmp ah,1
    jnz Ch2
    jmp getch

Ch2:
    cmp ah,2
    jnz Ch3
    jmp putch

Ch3:
    cmp ah,3
    jnz Ch4
    jmp puts

Ch4:
    cmp ah,4
    jnz Ch5
    jmp gets

Ch5:
    cmp ah,5
    jnz Q
    jmp scanf

Q:
    iret;最后返回
    jmp $
```

根据上述的类似于Switch语句的选择之后，则可以根据相应的功能号跳转至相应的程序段进行执行，每个程序段执行结束之后利用iret指令返回。

如下以getch子程序段为例：

```
getch:
    mov ax,cs;置其它寄存器与CS相同
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = CS
    push bp
    push bx
    mov bp,sp
    mov bx,[bp+12];char/ip1/ip2/cs/psw/bp/bx
    mov ah,0 ;功能号
    int 16h
    mov byte ptr [bx],al ;用一个字符指针保存输入的字符,Inch为C模块的变量
    mov sp,bp
    pop bx
    pop bp
    jmp near ptr Restart
    iret;一定要记住要iret指令返回!
```

如上所述即可以实现系统调用的功能，特别注意地就是在通过系统调用方式执行子程序段时，在栈中利用bp寄存器访问栈中数据时，要特别注意此时由于中断调用，因此被中断程序的程序状态字、偏移地址以及段地址均在栈中，因此访问时需要考虑这三个数据。否则无法正确访问参数值。

○ 修改21h的中断向量表

类似于实验四的编写中断过程，我们需要在程序开始时修改对应中断在中断向量表中的中断向量，根据实验四所学的知识可知，中断向量在0~1023处，每个中断含两个字，对应的中断向量位置为中断号×4处，分别将中断处理程序的偏移地址与段地址赋值给对应的中断向量处即可。具体的代码如下：

```
xor ax,ax
mov es,ax;清零段地址，中断向量在内存：0~1023处
mov word ptr es:[33*4],offset int21h;中断号21h*4
mov word ptr es:[33*4+2],cs
```

根据上述这两个分析以及步骤即可实现系统调用功能，通过功能号判断进入哪一个子程序段即可，实现一个中断向量的充分利用。

● 新增用户程序test.com

根据实验要求，需要新增一个用户程序用以测试上述封装好的且用系统调用执行的C库输入、输出函数，本人根据设计的六个输入输出函数：getch、putch、gets、puts、printf、scanf设计以下的测试方法：

即根据提示性的语句依次getch输入一个字符、putch输出一个字符；getes输入一个字符串、puts输出一个字符串；scanf输入一个字符，putch输出一个字符；scanf输入一个字符串，printf输出一个字符串。总共四个步骤的测试，完成了对所有自己封装Cku输入输出函数的调用测试。

对于C库的输入输出函数的测试是由C模块test.c编写的，编写一个指定C模块的入口模块enter.asm，再将test.c与以及include含由int 21h系统调用执行的C库输入输出子程序段的文件libs.asm的enter.asm链接编译成一个新增的test.com程序即可。

下面展示局部测试代码：

根据输出显示的提示信息，输入字符、字符串即可。

```

printf("Input a char and output a char\r\n");
getch(&Ch);
putch(Ch);
putch(' ');
putch(Ch);
putch(13);
putch(10);
printf("Input a str and output a str\r\n");
gets(s);
putch('\r');
putch('\n');
puts(s);
putch('\r');
putch('\n');
printf("Input a char and output a char\r\n");
scanf("%d", &Ch);
putch(Ch);
putch(' ');
putch(Ch);
puts("\r\n");
printf("Input a str and output a str\r\n");
scanf("%s", s);
puts("\r\n");
printf(s);
puts("\r\n");
printf("Input quit to exit\r\n");
gets(s);

```

而对于新的用户程序test.com，由于test.com的大小在编译时发现已经大于一个扇区大小因此在内核程序加载测试程序时，需要加载两个扇区。根据上述的操作，即可以设计出一个用以测试封装的C库输入输出函数的用户程序。程序运行效果将在实验运行结果部分展示。

• INT 22H中断处理程序设计

根据实验要求，需要设计22h中断的中断处理程序，用以在屏幕某个位置显示“INT22H”信息。基于此要求很自然地联想到利用BIOS的10h号中断显示字符串的功能调用进行显示，此中断调用在前几次实验中已经熟练地多次使用了。由于要求规定，每次触发22h中断就需要显示字符串，但由于对于中断的响应程序执行时间很短，因此为了方便观察字符串的显示效果，通过查阅BIOS的中断大全，了解到BIOS的15h号中断号的86h功能号的中断调用可以进行延时操作，在显示字符串后，调用此中断进行延时1s的操作，便于显示观察。

且为了区别每一次触发中断的响应过程，在显示字符串时，先进行了延时1s的操作，1s之后调用清除屏幕内容的功能调用进行该位置字符串的清除操作，同时对于每一次的中断响应，在显示字符串时进行了变色操作，设置一个color变量存储字符的显示属性，初值为1，每次响应之后color自增1，当color值为8时，重新置1，实现字符串显示的颜色变化循环。

首先，对于显示字符串，调用BIOS的10h显示字符串的功能调用：

```

mov ax,cs
mov es,ax;段地址
mov ah,13h ;功能号
mov al,0
mov bl,byte ptr es:[color];显示属性
mov bh,0
mov dh,12;显示字符串的行坐标
mov dl,35;显示字符串的列坐标
mov bp,offset Info ;"OUCH!OUCH!"字符串的偏移地址

```

```

mov cx,11;要显示的字符串长度
int 10h
inc byte ptr es:[color];颜色变量自增
mov al,8
cmp al,byte ptr es:[color];判断变量是否为8
jz Reset;若为8重置为1
jmp Dump;不为, 延时

```

显示完字符串之后, 进行延时操作, 延时2s:

```

mov ah,86h ;BIOS的15h中断号86h功能号的延时功能
mov cx,0Fh ;CX: DX= 延时时间(单位是微秒), CX是高字, DX是低字
mov dx,4240h ;1s=1000000us=0x0F4240
int 15h

```

延时显示之后, 需要调用BIOS的10h号中断清除该位置的字符串, 便于下一次的显示:

```

mov ah,6
mov al,0
mov ch,12 ;清屏的左上角坐标
mov cl,35
mov dh,12 ;清屏的右下角坐标
mov dl,45
mov bh,7 ;默认属性, 黑底
int 10h

```

最后, 再利用iret指令返回。

• MyOS模块组织

在本次实验中, MyOS的整个模块结构变得更加丰富了, 具体有以下几部分:

1. boot模块: 含boot.asm。

作用: 引导扇区程序, 用以加载操作系统内核, 并将系统控制器交给内核

2. kernel模块: 含rukout.asm kernalt.c basict.asm, PCB.h

作用如下:

rukout.asm: 用以修改21h、22h以及时间中断的中断向量, 并调用内核程序的入口——cmain函数。

kernalt.c: 用以组织整个内核程序, 主导系统的命令输入、程序执行操作, 根据底层汇编模块封装相关函数, 如: 输入输出字符串函数、字符串比较函数等等。

basict.asm: 汇编语言编写的底层功能实现模块, 包含清屏、输入一个字符、输出一个字符、读取扇区加载程序、时钟中断服务程序、**22h中断**的服务程序以及**Save**: 中断处理的保护现场、**Restart**: 中断处理的恢复现场函数。

PCB.h: 用以声明中断处理时的保护现场以及恢复现场所需的数据结构, 含进程控制块PCB, 进程表项以及保护现场的Save相关寄存器值等函数。

3. UP模块: 含UP1t.com~UP4t.com、test.com

作用: 含四个用户程序, 用以执行字符运动。

新增test.com用户程序, 由enter.asm,test.c,libs.asm三部分链接而成。

其中enter.asm指明test程序的入口；test.c函数用以调用汇编库中的相关输入输出函数用以文件测试；libs.asm用以封装多个C库的输入输出函数，并用int 21h的系统调用执行这些函数。

4. 表格模块：含tablet.com

作用：显示四个用户程序相关的信息，包括所在扇区、地址以及文件大小

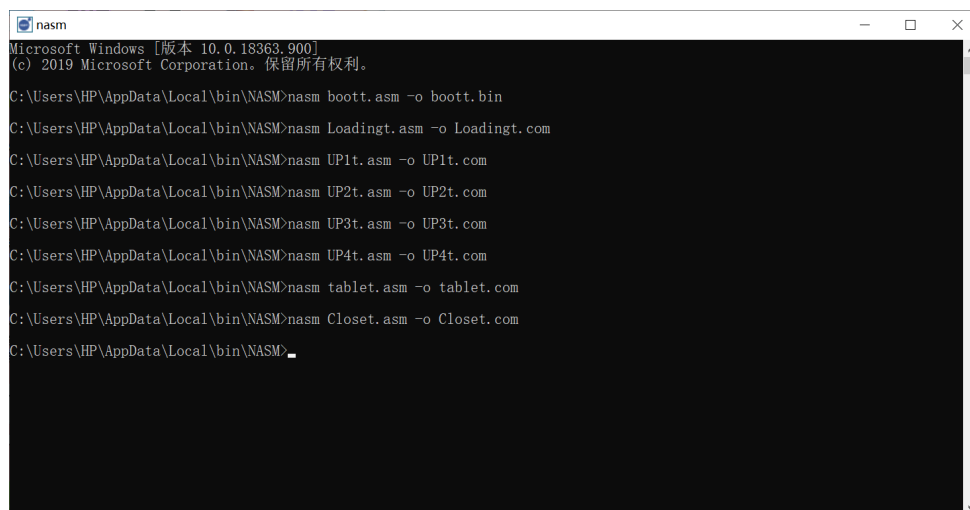
5. 开关机界面模块：含Loadingt.com Closet.com

作用：用以系统启动和退出时的界面显示。

• 实验操作过程

1. NASM编译

打开NASM，在框内输入指令，对于引导扇区程序boott.asm、四个用户程序UP系列、新增testC库输入输出函数测试用户程序，表格程序table.asm以及开关机界面显示Loading、Close程序，除了boott.asm引导扇区程序，其余我将他们都编译成COM格式。因此输入nasm+输入文件名+-o+输出文件名即可。生成同名的二进制文件。



```
Microsoft Windows [版本 10.0.18363.900]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\Users\HP\AppData\Local\bin\NASM>nasm boott.asm -o boott.bin
C:\Users\HP\AppData\Local\bin\NASM>nasm Loadingt.asm -o Loadingt.com
C:\Users\HP\AppData\Local\bin\NASM>nasm UP1t.asm -o UP1t.com
C:\Users\HP\AppData\Local\bin\NASM>nasm UP2t.asm -o UP2t.com
C:\Users\HP\AppData\Local\bin\NASM>nasm UP3t.asm -o UP3t.com
C:\Users\HP\AppData\Local\bin\NASM>nasm UP4t.asm -o UP4t.com
C:\Users\HP\AppData\Local\bin\NASM>nasm tablet.asm -o tablet.com
C:\Users\HP\AppData\Local\bin\NASM>nasm Closet.asm -o Closet.com
C:\Users\HP\AppData\Local\bin\NASM>
```

2. TCC+TASM+TLINK链接编译

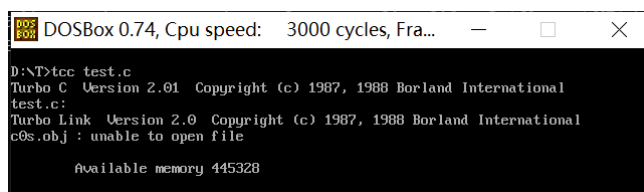
在DosBox中链接编译内核程序：rukout.asm、kernalt.c、bacist.asm、PCB.h

除此之外，还需链接编译新增test程序：Stuct.h、enter.asm、test.c、libs.asm

下面以test.com程序为例：

■ TCC编译

用tcc+.c 形式生成同名.obj文件：



```
DOSBox 0.74, Cpu speed: 3000 cycles, Fra...
D:\T\tcc test.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
test.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

Available memory 445328
```

■ TASM编译

用tasm+.asm形式先生成同名.obj文件，由于enter.asm包含了libs.asm文件，因此还需要用tasm+enter,obj形式再编译一次：


```
DOSBox 0.74, Cpu speed: 3000 cycles, Fra...
D:\T>tasm enter.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file: enter.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 463k

D:\T>tasm enter.obj
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file: enter.ASM to obj.OBJ
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 463k

D:\T>a
```

■ TLINK链接

用tlink /t /3 .obj .obj, .com形式生成test的COM格式文件：

```
DOSBox 0.74, Cpu speed: 3000 cycles, Fra...
D:\T>tlink /t /3 enter.obj test.obj, test.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

D:\T>
```

3. winHex工具编辑软盘

不同于实验一，在Linux环境使用dd命令进行1.44MB虚拟软盘映像的生成，在实验二、三、四中，我利用WinHex编辑软盘，两者相比较，我认为WinHex工具更为简便操作。因此，本次实验依然使用WinHex编辑软盘。

首先生成一个1.44MB的空白软盘myos5.img.利用winHex工具打开上述生成的各个二进制文件，先将“boot.bin”二进制文件内容拷贝到空白软盘首扇区，然后将内核程序kernel.com存放在扇区号为2开始的六个扇区空间，起始地址为200h；接着再从第19个扇区开始，依次存放四个用户程序(UP1~4)、表格程序、Loading程序、关机界面程序Close.com以及新增的测试程序test.com。第一个用户程序UP1存放在第19个扇区（2400H~2600H），第二个用户程序UP2存放在第20个扇区（2600H~2800H）的规则，以此类推将UP1~UP4的二进制文件以及表格程序、Loading程序、Close程序和test程序拷贝到对应扇区中。

拷贝操作的过程如下所示：

KERNEL.COM	Loading.com	tablet.com	boot.bin	TEST.COM	UP1.com	UP2.com	UP3.com	UP4.com	Close.com	cimg						
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	33	C0	8E	C0	26	C7	06	20	00	E2	02	26	8C	0E	22	00
00000010	26	C7	06	88	00	75	03	26	8C	0E	8A	00	8C	C8	8E	D8
00000020	8E	C0	8E	D0	BC	FF	FF	B4	02	B7	00	BA	00	00	CD	10
00000030	E8	21	00	E8	8C	08	EB	FE	55	8B	EC	56	57	1E	C5	76
00000040	06	C4	7E	0A	FC	D1	E9	F3	A5	13	C9	F3	A4	1F	5F	5E
00000050	5D	CA	08	00	50	53	51	52	B4	06	B0	00	B5	00	B1	00
00000060	B6	18	B2	4F	B7	07	B3	00	CD	10	B4	02	B7	00	BA	00
00000070	00	CD	10	5A	59	5B	58	C3	B4	00	CD	16	2E	A2	1C	11
00000080	C3	50	1E	06	55	33	C0	8E	C0	26	C7	06	80	00	BB	01
00000090	26	8C	0E	82	00	8B	EC	8C	C8	8E	C0	BB	00	A1	B4	02
000000A0	B0	01	B2	00	B6	01	8B	4E	0A	B5	00	80	E9	30	CD	13
000000B0	C7	07	00	01	C7	47	02	00	12	FF	2F	E8	73	00	8C	C8
000000C0	8E	D8	8E	C0	8E	D0	5D	07	1F	5D	07	1F	58	C3	E8	A6
000000D0	00	50	1E	06	55	33	C0	8E	C0	26	C7	06	80	00	0B	02
000000E0	26	8C	0E	82	00	8B	EC	8C	C8	8E	C0	BB	00	B1	B4	02
000000F0	B0	04	B2	00	B6	01	8B	4E	0A	B5	00	80	E9	30	CD	13
00000100	C7	07	00	01	C7	47	02	00	13	FF	2F	E8	73	00	8C	C8
00000110	8E	D8	8E	C0	8E	D0	C3	E8	5D	00	55	8B	EC	8B	46	04
00000120	B4	0E	B3	00	CD	10	8B	E5	5D	C3	CD	22	C3	00	00	00
00000130	00	16	50	53	51	52	54	55	56	57	1E	06	0F	A0	0F	A8
00000140	8C	C8	8E	C0	8E	D8	E8	08	02	0F	A9	0F	A1	07	1F	5F
00000150	5E	5D	5C	5A	59	5B	58	17	8C	C8	8E	C0	8E	D8	58	55
00000160	BD	2F	02	89	46	00	5D	E8	E3	02	55	BD	2F	02	8B	46
00000170	00	5D	50	E8	15	03	C3	8C	C8	8E	C0	8E	D8	E8	56	01
00000180	8B	E8	3E	8E	56	00	3E	8B	66	10	3E	83	7E	20	00	75
00000190	4C	E8	1F	03	E8	44	03	3E	FF	76	1E	3E	FF	76	1C	3E
000001A0	FF	76	1A	3E	FF	76	02	3E	FF	76	04	3E	FF	76	06	3E
000001B0	FF	76	08	3E	FF	76	0A	3E	FF	76	0C	3E	FF	76	0E	3E
000001C0	FF	76	12	3E	FF	76	14	3E	FF	76	16	3E	FF	76	18	58
000001D0	59	5A	5B	5D	5E	5F	1F	07	0F	A1	0F	A9	CF	83	C4	12
000001E0	EB	AF	50	53	51	52	55	06	1E	8C	C8	8E	C0	26	FE	0E

上图为“kernel”内核程序二进制文件部分内容，利用操作：选中需要复制内容，右键——编辑——复制选块——正常，完成复制操作。

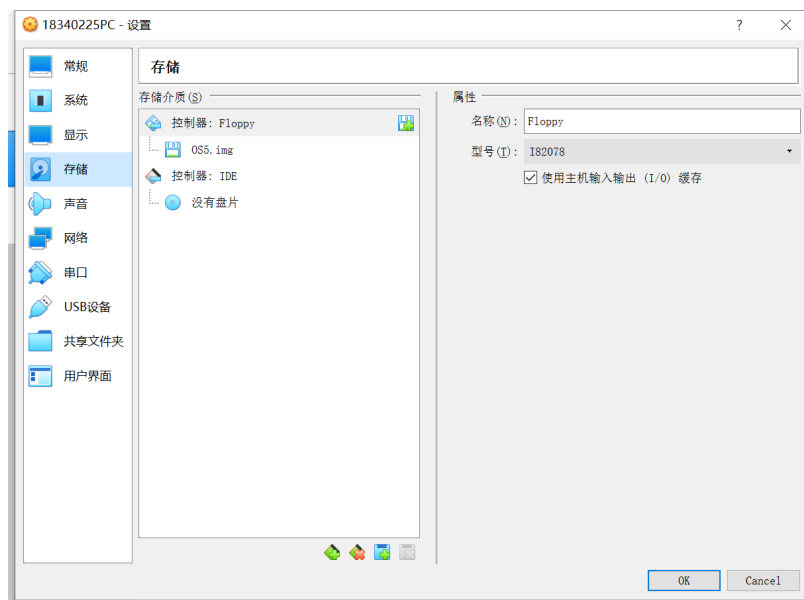
KERNEL.COM	Loadingt.com	tablet.com	boott.bin	TEST.COM	UP1t.com	UP2t.com	UP3t.com	UP4t.com	Closet.com	c.img							
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000200	33	C0	8E	C0	26	C7	06	20	00	E2	02	26	8C	0E	22	00	3AŽA&Ç a &E "
00000210	26	C7	06	88	00	75	03	26	8C	0E	8A	00	8C	C8	8E	D8	6Ç ~ u 6E Š 6EŽ0
00000220	8E	C0	8E	D0	BC	FF	FF	B4	02	B7	00	BA	00	00	CD	10	ŽAŽD4yy' . ° í
00000230	E8	21	00	E8	8C	08	EB	FE	55	8B	EC	56	57	1E	C5	76	è! 6E 6pUc1VW Áv
00000240	06	C4	7E	0A	FC	D1	E9	F3	A5	13	C9	F3	A4	1F	5F	5E	Ä~ üÑ6v É6n ^
00000250	5D	CA	08	00	50	53	51	52	B4	06	B0	00	B5	00	B1	00	jÈ PSQR' ° µ ±
00000260	B6	18	B2	4F	B7	07	B3	00	CD	10	B4	02	B7	00	BA	00	Ÿ ²0. ³ í ' . °
00000270	00	CD	10	5A	59	5B	58	C3	B4	00	CD	16	2E	A2	1C	11	í ZY(XA' í .ç
00000280	C3	50	1E	06	55	33	C0	8E	C0	26	C7	06	80	00	BB	01	ÄP U3AŽA&Ç e »
00000290	26	8C	0E	82	00	8B	EC	8C	C8	8E	C0	BB	00	A1	B4	02	6E , <l6EŽA» i'
000002A0	B0	01	B2	00	B6	01	8B	4E	0A	B5	00	80	E9	30	CD	13	° ² Ÿ <N µ 660í
000002B0	C7	07	00	01	C7	47	02	00	12	FF	2F	E8	73	00	8C	C8	Ç ÇG y/és 6E
000002C0	8E	D8	8E	C0	8E	D0	5D	07	1F	5D	07	1F	58	C3	E8	A6	Ž0ŽAŽD] j XAè;
000002D0	00	50	1E	06	55	33	C0	8E	C0	26	C7	06	80	00	0B	02	P U3AŽA&Ç e
000002E0	26	8C	0E	82	00	8B	EC	8C	C8	8E	C0	BB	00	B1	B4	02	6E , <l6EŽA» ±'
000002F0	B0	04	B2	00	B6	01	8B	4E	0A	B5	00	80	E9	30	CD	13	° ² Ÿ <N µ 660í
00000300	C7	07	00	01	C7	47	02	00	13	FF	2F	E8	23	00	8C	C8	Ç ÇG y/è# 6E
00000310	8E	D8	8E	C0	8E	D0	C3	E8	5D	00	55	8B	EC	8B	46	04	Ž0ŽAŽDÄè] U<i<F
00000320	B4	0E	B3	00	CD	10	8B	E5	5D	C3	CD	22	C3	00	00	00	' ³ í <ä]Äi"A
00000330	00	16	50	53	51	52	54	55	56	57	1E	06	0F	A0	0F	A8	PSQRTUVW "
00000340	8C	C8	8E	C0	8E	D8	E8	08	02	0F	A9	0F	A1	07	1F	5F	6EŽAŽ0è © i
00000350	5E	5D	5C	5A	59	5B	58	17	8C	C8	8E	C0	8E	D8	58	55	^]\ZY[X 6EŽAŽ0XU
00000360	BD	2F	02	89	46	00	5D	E8	E3	02	55	BD	2F	02	8B	46	¼/ %F jèa U¼/ <F
00000370	00	5D	50	E8	15	03	C3	8C	C8	8E	C0	8E	D8	E8	56	01	jPè Ä6EŽAŽ0èV
00000380	8B	E8	3E	8E	56	00	3E	8B	66	10	3E	83	7E	20	00	75	<è>ŽV ><f >f~ u
00000390	4C	E8	1F	03	E8	44	03	3E	FF	76	1E	3E	FF	76	1C	3E	Lè èD >yv >yv
000003A0	FF	76	1A	3E	FF	76	02	3E	FF	76	04	3E	FF	76	06	3E	yv >yv >yv >yv
000003B0	FF	76	08	3E	FF	76	0A	3E	FF	76	0C	3E	FF	76	0E	3E	yv >yv >yv >yv
000003C0	FF	76	12	3E	FF	76	14	3E	FF	76	16	3E	FF	76	18	58	yv >yv >yv >yv
000003D0	59	5A	5B	5D	5E	5F	1F	07	0F	A1	0F	A9	CF	83	C4	12	YZ[]^ i ©IfÄ
000003E0	EB	AF	50	53	51	52	55	06	1E	8C	C8	8E	C0	26	FE	0E	è PSQRU 6EŽA&p

上图为1.44MB软盘，将上述“kernel”的内容复制到此软盘的首扇区，操作为：右键——剪贴板数据——粘贴。

其余程序的拷贝操作类似。

4. 软盘启动裸机

将上述生成的引导扇区程序二进制文件、内核COM格式文件、用户程序以及表格程序、Loading程序、新建的test程序以及关节界面程序Close,依次按照扇区分配复制到1.44MB软盘的对应位置并保存后，将1.44MB软盘作为引导盘启动虚拟机。



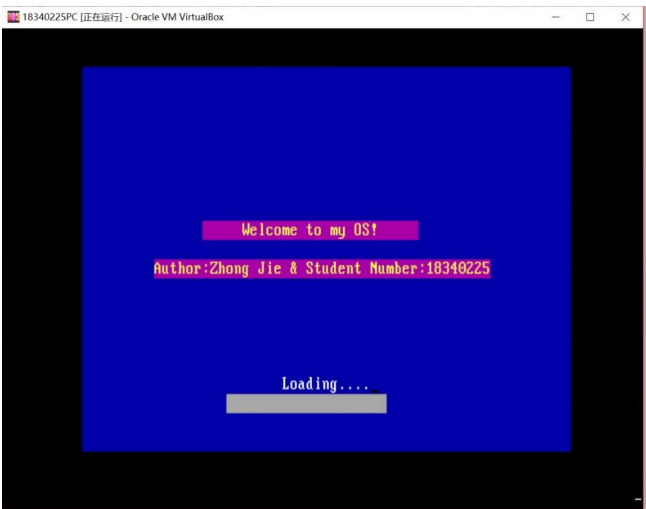
• 实验运行结果

成功启动虚拟机之后，虚拟机就会将程序加载到虚拟机的内存空间中运行，可以看到Loading程序的启动界面，也可以观察到内核程序的界面，接着输入help命令，依次按照菜单，输入选择运行不同的命令如：UP1~UP4、dir、ls、batch、INT、quit，执行不同的用户程序，显性调用用户自定义中断21h~24h、查看用户程序信息、列出系统文件组织等功能，用户子程序的运行均以延时

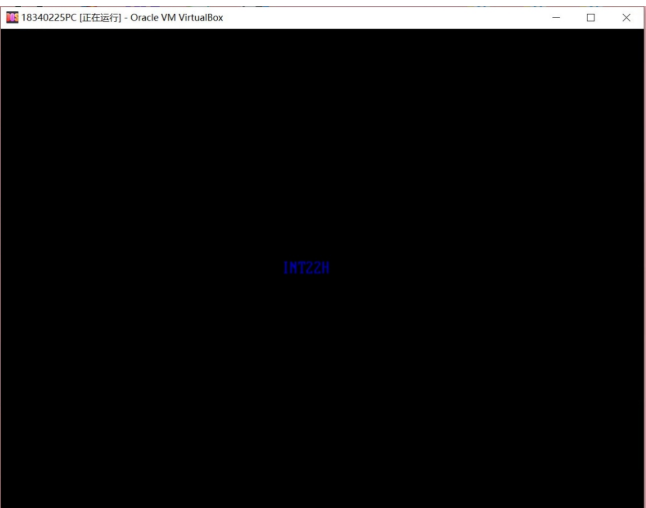
退出的方式返回到内核程序。可以看到时钟中断响应程序的风火轮效果，而且可以测试在用户程序运行期间触发键盘中断的显示效果。

程序运行结果如下：

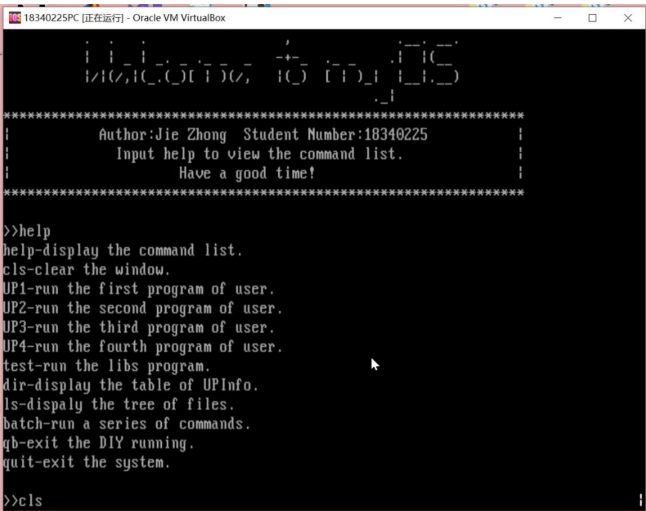
Loading程序加载界面：



进入内核界面，显性直接调用int 22h中断，显示“INT22H”字符串：



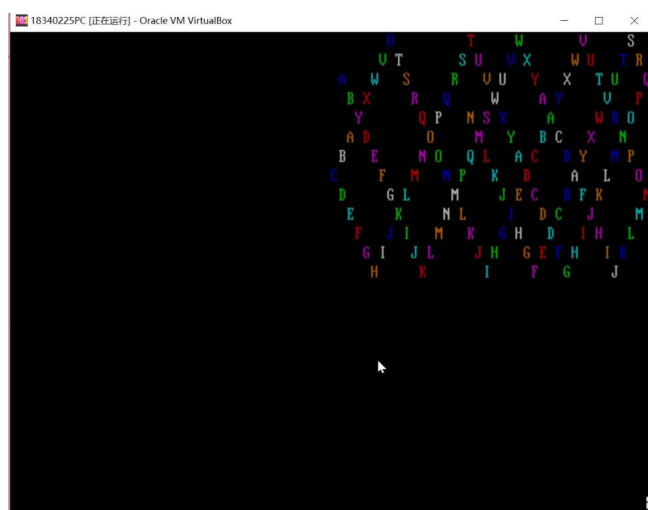
内核界面，输入“help”，显示命令清单：



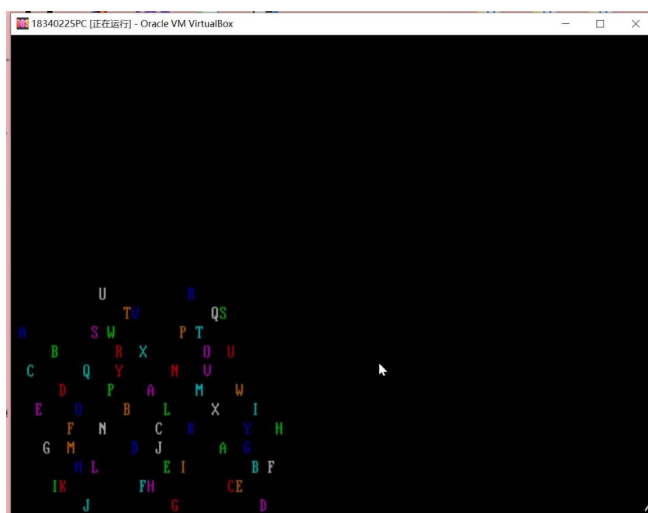
输入“cls”,系统执行清屏操作，接着输入“UP1”，显示第一个用户程序：



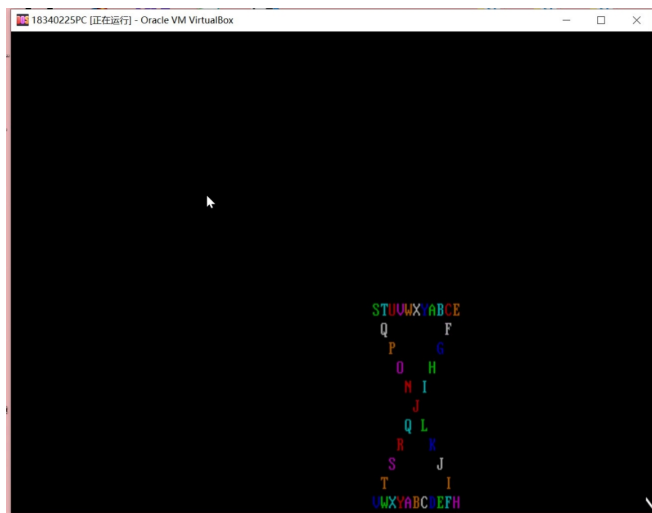
延时回到内核程序，接着输入“UP2”，显示第二个用户程序：



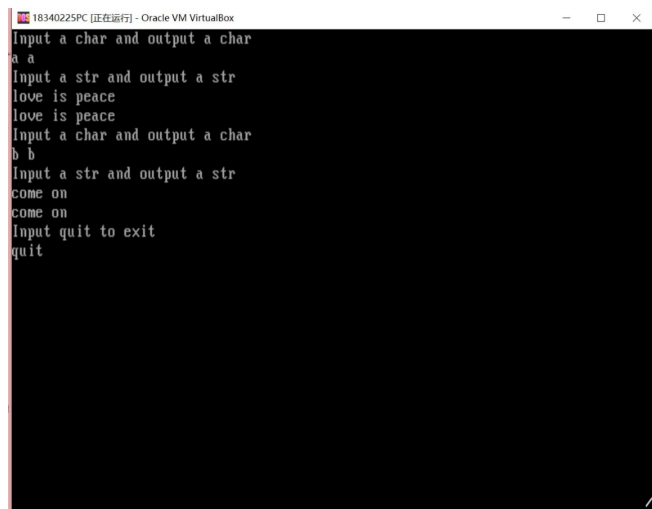
延时回到内核程序，接着输入“UP3”，显示第三个用户程序：



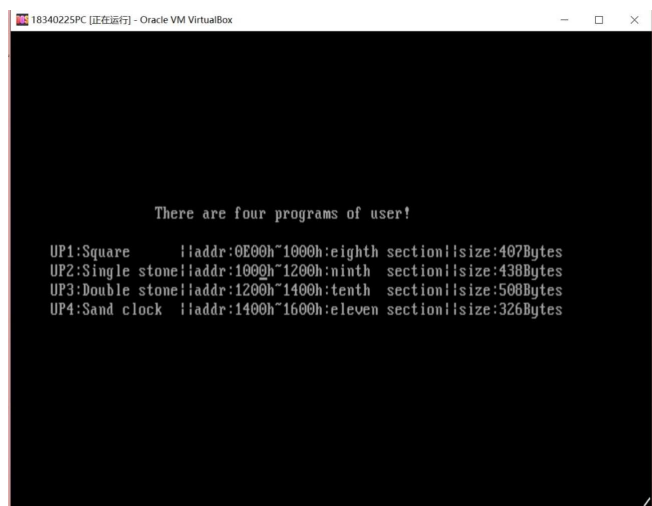
延时回到内核程序，接着输入“UP4”，显示第四个用户程序：



延时回到内核程序，接着输入“test”，显示新增的用以测试C库输入输出设计的用户程序：



quit命令回到内核程序，接着输入“dir”，查看显示四个用户程序的表格信息：



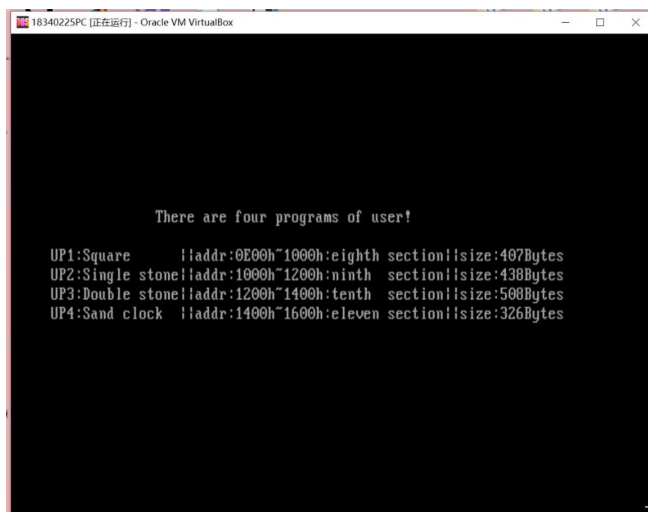
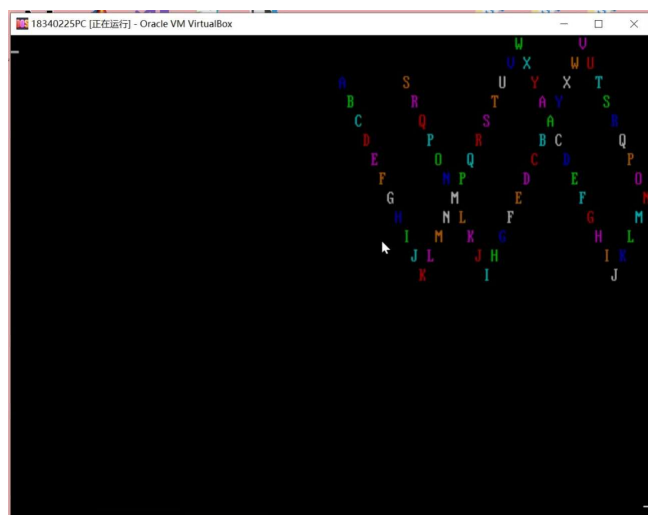
延时回到内核程序，接着输入“ls”，查看此简易系统的文件组织：

```
18340225PC [正在运行] - Oracle VM VirtualBox
>>ls
!-boot.bin
!-kernal.com
!-----rukou.asm
!-----PCB.h
!-----kernal.c
!-----bacis.asm
!-----UP1.com
!-----UP2.com
!-----UP3.com
!-----UP4.com
!-----test.com
!-----enter.asm
!-----Stuct.h
!-----test.c
!-----libs.asm
!-----table.com
!-----Loading.com
!-----Close.com
>>
```

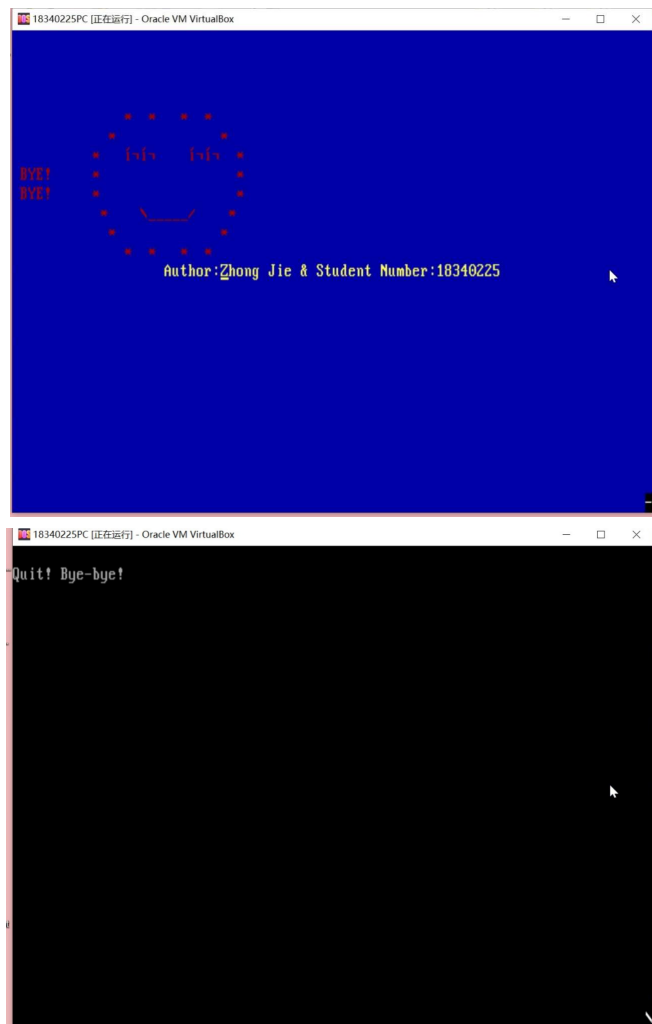
再输入“batch”，测试批处理命令，执行一批“UP3、UP2、dir、UP4、qb”的指令：

```
18340225PC [正在运行] - Oracle VM VirtualBox
Please separate cmds by Spaces.
UP3 UP2 dir UP4 qb
```

```
18340225PC [正在运行] - Oracle VM VirtualBox
U
R
TV
QS
A S W P T
B R X O U
C Q Y N U
D P A M W
E O B L X
F M C K A Y
G H L B J A B
I K F H E I C
J G
```



输入“quit”退出系统，显示关机界面与提示语句。



至此，完成所有功能测试，实验运行结果与预期相符合。

实验心得

本次实验是主要涉及三个部分的内容：C库输入、输出过程封装设计，22h中断处理程序设计、21h号的系统调用的设计以及中断处理的保护现场Save、恢复现场Restart过程设计。对于我而言，由于在实验三的内核程序设计的过程中已经初步实现了C库相关的gets、printf过程，因此在本次实验中，对于C库的输入输出过程的设计相对比较熟悉，只需要在实验三已经封装好的输入输出字符串函数的基础上增加输入、输出字符函数，并且结合本次实验的实验要求做出相应修改即可；而对于22h的中断处理程序的设计，由于在实验四中我们主要的实验任务就集中在中断处理程序的设计，因此对于中断的处理程序的设计以及修改中断向量表的操作相对熟悉；本次实验的重难点是系统调用的设计以及中断处理的保护现场过程Save、恢复现场过程Restart的设计。

相比之下，21h的系统调用设计还是相对简单的，结合在实验课上老师详细讲述的系统调用的相关知识，再加上自己的理解，其实系统调用相对于一般的中断调用就是提供了一个服务程序的总入口，然后在中断入口处根据设定的功能号对应表，根据功能号进行匹配，找到功能号对应的子程序段，开始执行，执行完成之后利用iret指令返回。使用系统调用极大提高了中断调用的利用率，通过一个总入口进入，再根据功能号选择子程序执行提高效率。单从系统调用的设计思想来看，是很容易理解，汇编编程的工作量也不大。但在编程完毕测试过程中，我却遇到了不少问题，一个大问题就是，我只是将原本独立的过程搬进中断处理程序中，以一个子程序段的形式出现，并未做大改动，但原本运行正常的输入输出过程，却一直无法正确输入、正确输出，我利用Bochs工具进行调试，当我用xp命令查看传入参数的内容时，发现输入的字符并没有成功保存在我传入的参数中，我感到很疑惑。后来，我利用print-stack命令查看传入参数后，进入系统调用执行子程序段的栈中内容，才发现原来是通过int 21h的系统调用进入子程序段执行，取出参数时，由于中断调用会被被中断程序中中断处的程序状态字、程序偏移量以及段地址入栈，因此我们在利用sp寄存器的值，获取栈中参数的时候应该要考虑栈中还有上述的三个数据，

因此不可以直接套用之前实验设计的代码，应该在取得参数的位置上做出适当的修改。这虽然是一个难度不大的问题，但却是很细节的问题，稍不注意就会使得程序出错且难以发觉。

当然，本次实验对于我来说，最为挑战的是中断处理过程中的保护现场Save、恢复现场Restart过程的设计。最开始看到这个实验要求时，脑子里只浮现出最简单的想法，将执行中断处理程序前的所有寄存器的值入栈，然后再出栈，显然这是不可行的，在执行中断处理程序时ss、sp的值会发生改变，因此我们很难正确访问到中断程序执行前的栈中内容。好在根据老师的提示，想到了可以利用C语言定义一些数据结构用以保存执行中断程序前的所有上下文寄存器的值，并在中断后利用Restart过程将数据结构中保存的数据再逐一恢复到所有寄存器中。设计数据结构的过程比较简单，只需要根据所有寄存器的类型进行编写即可，Save过程则只需要调用一个C编写的保护寄存器值的函数，依次传入正确的参数即可。

在实验中，我遇到的最大的困难就是Restart过程中的程序能够正确回到被中断点继续执行，最开始我一直遇到内核程序中断时无法正确Restart、用户程序也无法正确Restart的问题。后来利用Bochs单步调试发现，在Restart过程中，需要先将数据结构PCB中的ss寄存器、sp寄存器的值恢复，但此时sp寄存器的值，并不是正确的值，这通过将Save过程中的栈内容记录下来，再将Restart过程中恢复sp寄存器的值以后的内容记录下来比较可以发现，Restart过程中恢复的sp寄存器的值与正确的sp寄存器的值相差18，只要对于恢复后的sp寄存器的值加上18程序即可正确恢复并执行。刚开始我百思不得其解，为什么会相差18呢，后来发现是由于我们在调用C编写的保存上下文寄存器的函数时，通过栈传递参数的，每入栈一个数据，sp减2，而在sp入栈之前，已有9个数据入栈了，因此此时入栈的sp的值并不是被中断程序刚刚被中断时的值，需要加上 9×2 才是正确的值。明白了这个道理之后，我的Save和Restart过程便可以正确地执行了。

实验不是一件易事，总是在不断发现问题与解决问题。但收获也就是在发现与解决中积攒起来的，本人设计的简易操作系统仍还有很多欠缺，目前正在朝着更好的方向努力，希望可以进一步地提高自己的编程水平，努力向上！

参考文献

1. 《BIOS中断大全》：https://blog.csdn.net/weixin_37656939/article/details/79684611
2. 《突破512字节的限制》：<https://blog.csdn.net/cielozhang/article/details/6171783/>
3. 《X86汇编如何设置延时》：<https://blog.csdn.net/longintchar/article/details/70149027>
4. 《Bochs调试指令》：https://blog.csdn.net/ddna/article/details/4997695?utm_medium=distri_bute.pc_relevant.none-task-blog-baidujs-1