

Report

数据科学与计算机学院 18级超算方向 18340161 田蕊

一、实验题目

实现系统调用

二、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

三、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

四、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；int 22h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。

五、实验过程

(一)、实现save和restart过程

实现思路

save和restart过程的实质其实就是保护现场和回复现场。我们采取的策略是在内核程序的C部分添加一个结构变量保存我们想要保存的所有寄存器的值，每次保存现场的任务就是将内存中寄存器的值保存到结构变量中，恢复现场的任务就是把内存中结构变量的值写回寄存器中。需要注意的是ip寄存器，cs寄存器和程序状态字寄存器这单个变量要利用栈里面的值，而不能直接将正在执行的程序的变量放在内存中。因为我们要保存的不是save和restart过程中的程序执行的ip，cs和程序状态字，而是调用中断的程序的ip,cs和程序状态字。在程序调用中断的过程时，会将程序状态字，cs和ip自动压栈，所以我们所需要做的时在栈中取出数据。在恢复现场的时候，同样要将这三个值重新压栈，这样才能利用iret指令返回。

所以我们要将save和restart看成是一个统一的过程而不是分立的过程。当发生中断调用的时候，我们首先执行call _save指令，之后要在_save中跳转到中断控制程序，要注意这里必须要直接跳转而不能用ret指令。因为在这之前我们已经将cs，ip和程序状态字寄存器弹栈了，所以是不能调用ret指令的。在终端控制程序执行结束，在执行jmp指令跳转到_restart过程，需要注意的是这里也只能使用jmp，最后在_restart过程中将cs,ip和程序状态字寄存器重新压栈之后才能执行iret指令进行返回。

数据结构设计

```
typedef struct{
    int ax;
    int bx;
    int cx;
    int dx;
    int cs;
    int ds;
    int es;
    int ss;
    int sp;
    int bp;
    int di;
    int si;
    int ip;
    int flag;
} cpuRegister;
cpuRegister reg;
cpuRegister* preg = &reg;
```

保存寄存器内容的结构变量我们在C语言中定义，每一个寄存器的内容用一个int来表示。可以看见，在我们的程序中不仅定义了一个结构变量，还定义了一个结构变量的指针，实际上我们向_resave过程传递的参数是这个指针。我认为这一操作是合理的，因为在后续的实验中我们会向_save过程传递过程的cpuRegister变量，这个时候只要把变量地址赋给指针就可以了。

在内存中我们可以将一个结构变量看成是一个整体，在内存中是一个连续的存储单元。所以在汇编语言中只需要利用变量相对于preg地址的偏移就可以得到我们需要访问的地址。

代码说明

_save过程

```
_save:
    push ds
    push cs
    pop ds
    pop word[ds_save]

    pop word[ret_save]

    mov word[si_save], si
    mov si, preg

    pop word[si+48]; 保存ip
    pop word[si+16]; 保存cs
    pop word[si+52]; 保存flag

    mov word[si], ax
    mov word[si+4], bx
    mov word[si+8], cx
    mov word[si+12], dx
    push word[ds_save]
    pop word[si+20]
    mov word[si+24], es
    mov word[si+28], ss
    mov word[si+32], sp
    mov word[si+36], bp
    mov word[si+40], di
    push word[si_save]
    pop word[si+44]; si

    jmp word[ret_save]
```

在_save过程中，我们在汇编语言中还定义了几个辅助变量，分别是ds_save, ret_save, si_save。这三个变量分别保存ds寄存器的值，返回地址和si寄存器的值。因为在保存现场的过程中，我们要利用堆栈的内容，所以这些值我们不能简单地利用压栈来保存，这样会影响我们后续的弹栈。我们需要用到si寄存器来保存结构变量的首地址，所以在使用之前要把si寄存器中的内容先保存下来，利用的是这一条语句mov word[si_save], si。同时因为在_save过程的最后我们要用jmp语句跳转到中断处理程序，所以还要将返回地址保存到变量中使我们能够正确跳转，实现这一条要求的语句是这个pop word[ret_save]。我们从栈顶弹出返回地址，因为当时栈的形式是这样的：stack: psw/cs/pc/ret addr

在过程的最后我们利用保存的返回地址跳转回中断控制程序。

_restart

```
_restart:
    mov si, preg
    mov ax, word[si]
    mov bx, word[si+4]
    mov cx, word[si+8]
```

```

mov dx, word[si+12]
mov es, word[si+24]
mov ss, word[si+28]
mov sp, word[si+32]
mov bp, word[si+36]
mov di, word[si+40]

push word[si+52];flag压栈
push word[si+16];cs压栈
push word[si+48];ip压栈

push word[si+44]
push word[si+20]
pop ds
pop si

push ax
mov al,20h
out 20h,al
out 0A0h,al
pop ax

iret

```

在_restart过程中,我们的主要任务就是将结构变量中的内容写回寄存器中,这一操作十分简单我们就不再赘述。但是需要注意的是,为了能够正确的执行中断返回,我们的很重要的一个操作就是将程序状态字寄存器,cs寄存器和ip寄存器重新压栈。这样才能使栈顶和我们刚刚程序的时候保持一致从而正确执行中断返回指令。

(二)、内核增加int 20h、int21h和int 22h软中断的处理程序

相关知识

关于系统调用

所谓系统调用,就是由操作系统实现提供的所有系统调用所构成的集合即程序接口或应用编程接口(Application Programming Interface, API)。是应用程序同系统之间的接口。

系统调用原理

操作系统中的状态分为管态(核心态)和目态(用户态)。特权指令:一类只能在核心态下运行而不能在用户态下运行的特殊指令。不同的操作系统特权指令会有所差异,但是一般来说主要是和硬件相关的一些指令。访管指令:本身是一条特殊的指令,但不是特权指令。

(trap指令)。基本功能:“自愿进管”,能引起访管异常。

用户程序只在用户态下运行,有时需要访问系统核心功能,这时通过系统调用接口使用系统调用。

系统调用与中断的联系

在CPU内部产生,说这类中断是同步的,意思是中断信号的发射时间一定在当前指令执行结束之后。一般来自CPU的内部事件或程序执行中的事件,如非法操作码、地址越界、浮点溢出等。

Synchronous interrupts (异常)又分为以下若干类:

- **Processor-detected exceptions**: 处理器在执行指令时检测到的中断，如除零操作。
- **Faults**: 发生了某个异常条件，但异常条件被消除后，原来的程序流程可以继续执行而不受任何影响，如缺页异常。注意触发中断的指令会被重新执行。
- **Traps**: 由陷入指令引起的中断，通常用于程序调试。
- **Aborts**: CPU内部有重要错误发生，例如硬件错误或系统表值出现错误。一旦这种中断发生，错误将不可恢复，只能将当前进程终止。
- **Programmed exceptions**: 也称为 **software interrupts (软中断)**，由程序员的代码主动发起的中断，用来实现系统调用。如在Linux中，就是用`int 0x80`指令实现系统调用。

综上所述，系统调用是一种特殊类型的中断（软中断）。

实现思路

我们通过中断来实现系统调用，也就是说我们将系统调用的控制程序封装到内核的中断执行程序当中，这样程序只需要通过调用中断就可以执行我们定义好的程序，实现系统调用，同时还要为用户提供一个指令接口，来实现用户和系统的交互。所以我们的程序设计的整体结构如下所示。

```
_int22h:
    push ebp
    mov ebp, esp
    int 22h
    pop ebp
    ret
_int22hsys:
    cli
    call _save
    ;=====
    ;中断执行程序
    ;=====
    jmp _restart
```

之后我们还要在内核处理程序的最开始将`_int22hsys`装载到中断向量表中。这样当程序再调用中断的时候，就会执行我们先定义好的程序段。

除了上述说明的实现外，我们还可以在程序内部实现中断的不同功能号。只需要根据`al`寄存器的值实现一个`switch`跳转结构即可，具体实现方法如下所示

```
    cmp al, 0
    jz fno0
    cmp al, 1
    jz fno1
    cmp al, 2
    jz fno2
    cmp al, 3
    jz fno3
    jmp endsyscall
fno0:
fno1:
fno2:
```

```
fno3:
endsyscall:
```

实现功能

int21h

采用上述的方法，就可以实现不同的功能跳转，我们在这里实现的功能是fno0实现输出，fno1实现输入一个字符，fno2实现输入一个字符串，fno3将我们之前的反弹程序C封装到了其中。

in20h

实现从用户程序返回内核程序的功能。我们这里的实现方法十分的简单粗暴，就是直接将内核返回地址装载到我们所保存的结构体变量中，然后直接返回。事实上我认为这样的方法是不是很合理的，但是十分的有效orz.

in22h

按照要求in22h只需要实现简单的在屏幕上输出int22h即可，所以我们的实现方法也很简单，只要将字符放在现存的相应位置即可。

(三)、保留无敌风火轮显示，取消触碰键盘显示OUCH!

这个过程只需要简单的增删代码，没有难度，我们不再赘述。

(四)、设计C语言库函数并链接到测试程序

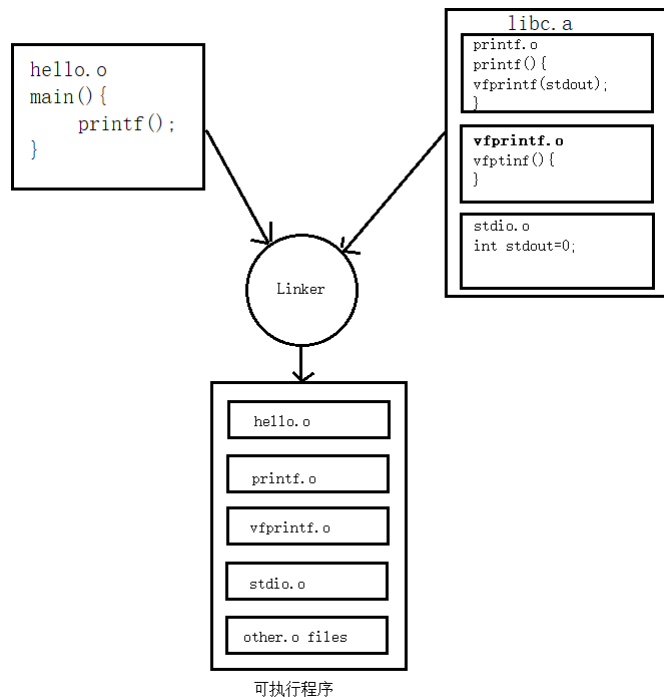
相关知识

这一部分主要的难点是在链接部分，我们所采用的编程环境是gcc+NASM，使用ld指令来进行链接，所以这里我们要想来介绍一下静态链接。

静态链接

在我们的实际开发中，不可能将所有代码放在一个源文件中，所以会出现多个源文件，而且多个源文件之间不是独立的，而会存在多种依赖关系，如一个源文件可能要调用另一个源文件中定义的函数，但是每个源文件都是独立编译的，即每个.c文件会形成一个.o文件，为了满足前面说的依赖关系，则需要将这些源文件产生的目标文件进行链接，从而形成一个可以

执行的程序。这个链接的过程就是静态链接



上图就是一个静态链接程序的示例。

实现思路

我们的测试程序是要用C语言直接完成一个main函数，但是我们之前写的用户程序都是用汇编来完成的，使得我们可以直接在程序的最开始通过汇编指令实现四段合一操作，但是如果只是用一个C语言main函数我们无法完成这个需求。这里采取的方法是借鉴内核程序的操作，就是在静态链接的时候，把库函数链接在用户程序的前面，然后在库过程的汇编部分执行四段合一后跳转到C语言函数部分，C语言main函数执行完毕，再跳回汇编库过程然后实现用户程序返回。

相信这样的操作的合理性会受到质疑，这样的操作是不是违背了用户程序和库函数的独立性呢？我认为这是没有问题的，因为我们这样设计库函数并不对用户程序有什么要求，只需要用户程序将库过程包含进去就可以了，虽然在库过程中进行了对main的跳转，但是这一跳转也不针对任何特定的程序，而是一个通用的过程，所以我认为是合理的。

我们在库函数中封装了puts过程，gets过程，_getC过程，putchar过程，strlen过程，在用户程序中我们对上面几个函数进行测试。注意，我们这里定义的puts函数自动输出换行。

测试程序内容如下所示

```
int main() {
    char* buf1 = "I am Tianrui hhhh";
    char* help1 = "please input a string: ";
    char* help2 = "buf1 is: ";
    char* help3 = "buf2 is: ";
    char* help4 = "length of buf2 is: ";
    char* help5 = "please input a char: ";
    char* help6 = "the char you have input is: ";
    char* help7 = "please input any key to continue";
    char newline = '\n';
    char buf2[100] = {0};
    char temp = 0;
```

```

int len = 0;

//测试输入gets函数和输出puts函数
puts(buf1);
puts(help1);
gets(buf2);

puts(help2);
puts(buf1);
puts(help3);
puts(buf2);

//测试strlen函数和putchar函数
puts(help4);
len = strlen(buf2);
len = len + 48;
putchar(len);
putchar(newline);

//测试_getC函数
puts(help5);
temp = _getC();
puts(help6);
_putC(temp, 7);
putchar(newline);

puts(help7);
_getC();

}

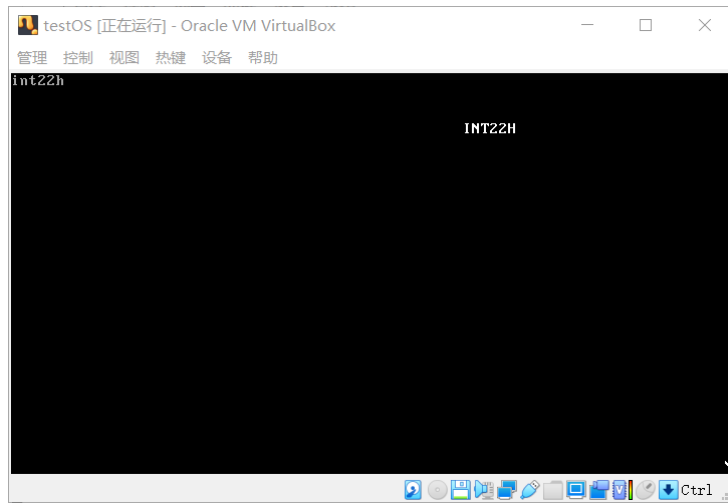
```

完成测试函数后，我们在内核中增加相应的接口，将这个测试程序编号为程序E，只需要在内核程序输入run e指令就可以运行这个程序。

六、实验结果

int22h执行效果



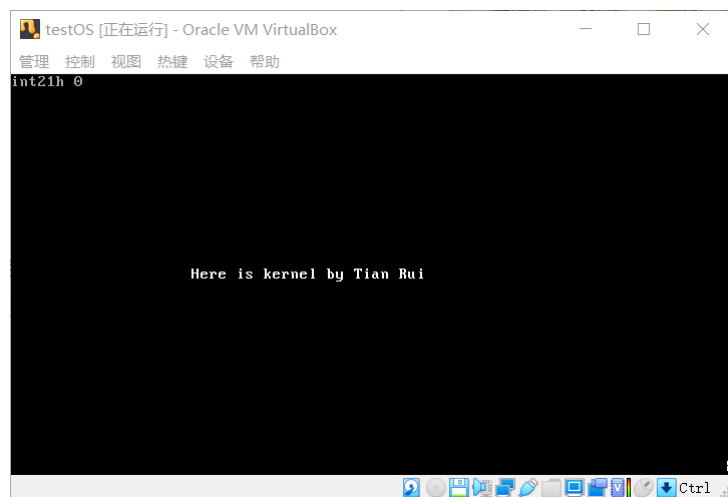


上图为int22h的执行效果，根据右下角风火轮的形状不同我们可以知到风火轮在执行过程中不断转动，具体效果也可以通过运行上交的软盘映像文件得到。

int21h执行效果

功能号0

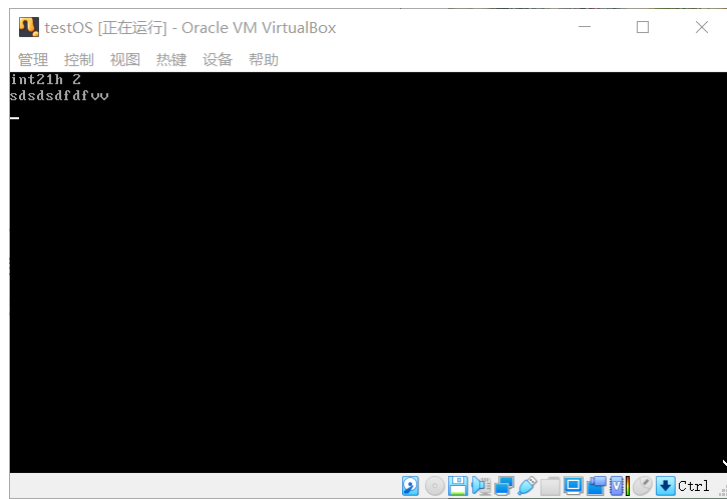
功能号0是输出功能，这里我们输出一条语句来表明功能号0的功能



功能号1

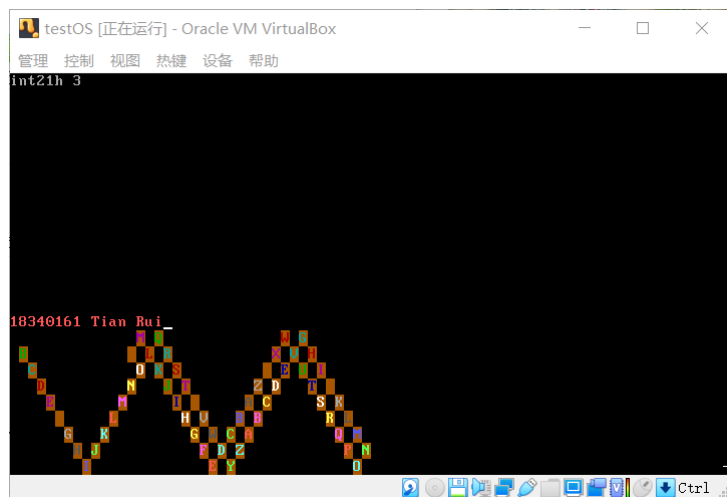
功能号2

功能号2的功能是输入一个字符串，首先输入指令int21h 2，之后输入我们的要输入的字符串，这个字符串就被保存到了一个目标地址中。



功能号3

功能号3实现的功能是将之前的反弹程序C封装到了中断中，所以输入int21h 3指令，可以看见反弹程序C在运行

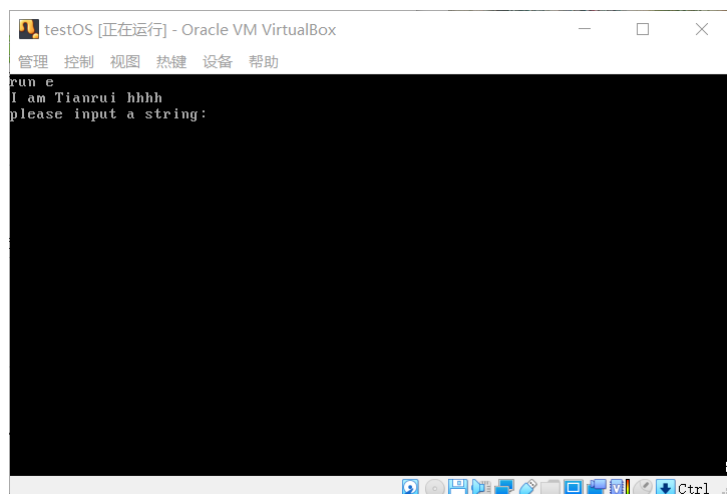


int20h执行效果

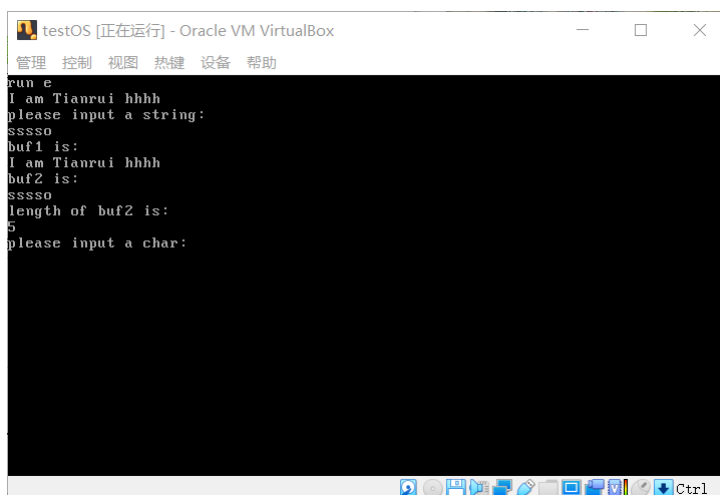
int20h的效果是用户程序结束是返回内核准备接受命令的状态，这一个过程没有办法在截图中展示出来，具体的执行效果可以参见上交的软盘映像文件执行效果。

测试程序运行效果

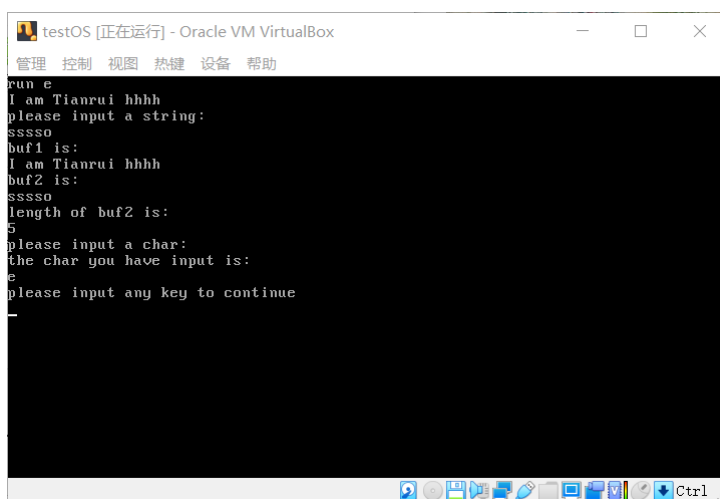
首先输入指令run e，程序被执行，输出程序中直接定义的第一个字符串和提示信息。



根据提示信息输入一个任意字符串后，可以看见程序输出了两个字符串，并且计算出了字符串的长度。运行结果如下所示



再根据提示信息输入一个字符后，可以看到运行结果如下所示



同时，根据右下角风火轮的斜杠位置不同可以知道这个过程中我们的风火轮是在一直转动的。

七、实验心得

1. 在保护现场和恢复现场的过程中，需要注意的是ip寄存器，cs寄存器和程序状态字寄存器这个变量要利用栈里面的值，而不能直接将正在执行的程序的变量放在内存中。因为我们要保存的不是save和restart过程中的程序执行的ip，cs和程序状态字，而是调用中断的程序的ip,cs和程序状态字。在程序调用中断的过程时，会将程序状态字，cs和ip自动压栈，所以我们所需要做的时在栈中取出数据。在恢复现场的时候，同样要将这三个值重新压栈，这样才能利用iret指令返回。
2. 在_save过程中我们最后利用保存的ret_save跳转到中断控制程序，在最开始的时候我采取的措施是这样的

```
mov ax, word[ret_save]
jmp ax
```

这样的操作程序可以正确跳转但是不能正确执行。后来经过调试发现是因为在这个过程中ax中的值就改变了，所以后来修改程序，直接执行下列语句

```
jmp word[ret_save]
```

这样就不会修改其他寄存器的内容实现直接跳转。

3. 在_save过程中将preg的内容写进si寄存器的时候我遇到了一些问题。我们期望的是si寄存器保存的是preg的内容，而不是reg的内容，这就涉及到我们究竟要使用mov si,preg语句还是mov si,[preg]，我们认为前者是合理的，虽然网上很多资料给了后者，但我认为第二条语句是不正确的，他只是能恰巧运行。在调试的过程中我们发现mov si,preg是将reg变量的地址放在si中，而mov si,[preg]是将reg的值放在si中，因为reg中的内容为0，所以会将我们所保存的寄存器内容保存在ds:0000h处，这个地方没有被使用到所以程序也能正确运行，但是我们要知道这其实是不对的。
4. 在实现内核功能号的时候，我们需要调用内核中的函数，如果我们在main中直接调用，那么这是一个远调用，但是在内核中这样就是一个近调用，但是这样一来参数就会出问题，所以要注意在调用函数的时候要先压入一个0字节，这样保证栈的一致性。具体代码如下

```
push eax
mov al, 'C'
movsx eax, al
push eax
push 0x00
call _call_proc
pop ax
pop eax
pop eax
jmp endsyscall
```

5. 在实验的过程中，调用了用户程序之后在bochs中可以正确执行，但是在Virtual Box中却不能正确执行，这个真的是不明所以orz。在bochs中程序能够正确返回，但是在virtual box中就不可以正确返回，最后真的心力交瘁，决定还是在程序的最后显式调用int 20h，让程序返回内核，这次问题就解决了。