Report

数据科学与计算机学院 18级超算方向 田蕊

一、实验题目

实现时间片轮转的二态进程模型

二、实验目的

1. 学习多道程序与CPU分时技术
2. 掌握操作系统内核的二态进程模型设计与实现方法
3. 掌握进程表示方法
4. 掌握时间片轮转调度的实现

三、实验要求

1. 掌握时间片轮转调度的实现
2. 扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。
3. 修改时钟中断处理程序，调用时间片轮转调度算法。
4. 设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。
5. 修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。
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

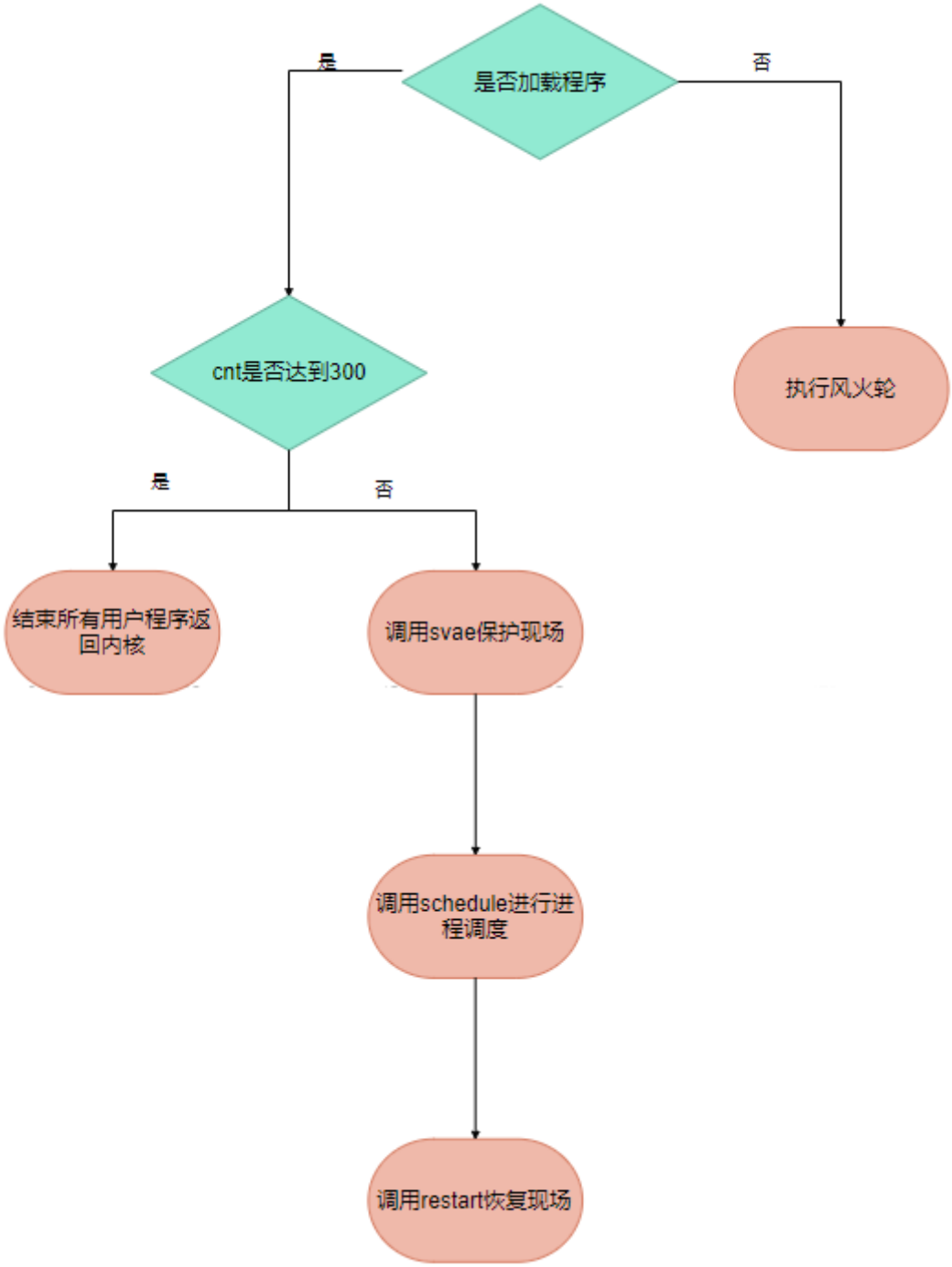
四、实验内容

1. 修改实验5的内核代码，定义进程控制块PCB类型，包括进程号、程序名、进程内存地址信息、CPU寄存器保存区、进程状态等必要数据项，再定义一个PCB数组，最大进程数为10个。
2. 扩展实验五的的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，再实现分时运行
3. 修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程
4. 内核增加进程调度过程：每次调度，将当前进程转入就绪状态，选择下一个进程运行，如此反复轮流运行。
5. 修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的运行。

五、实验方案

（一）、方案思想

在以前的原型操作系统顺序执行用户程序，内存中不会同时有两个用户程序，所以CPU控制权交接问题简单，操作系统加载了一个用户到内存中，然后将控制权交接给用户程序，用户程序执行完再将控制权交接回操作系统，一次性完成用户程序的执行过程。在时间片轮转的二态进程模型中，我们采用时钟中断打断执行中的用户程序实现CPU在进程之间交替。简单起见，我们让两个用户的程序均匀地推进，就可以在每次时钟中断处理时，将CPU控制权从当前用户程序交接给另一个用户程序。每一次时钟中断，中断服务程序就保存触发中断程序的现场，然后将程序转为就绪态，将下一个要执行的程序转换为运行态，然后恢复现场。在时钟中断内，程序的具体流程图如下。



（二）、相关原理

进程控制块PCB

进程控制块（Processing Control Block），是操作系统核心中一种数据结构，主要表示进程状态。其作用是使一个在多道程序环境下不能独立运行的程序（含数据），成为一个能独立运行的基本单位或与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。 PCB通常是系统内存占用区中的一个连续存区，它存放着操作系统用于描述进程情况及控制进程运行所需的全部信息，它使一个在多道程序环境下不能独立运行的程序成为一个能独立运行的基本单位或一个能与其他进程并发执行的进程。

PCB通常记载进程之相关信息，包括：

- 程序计数器：接着要运行的指令地址。
- 进程状态：可以是new、ready、running、waiting或 blocked等。
- CPU暂存器：如累加器、索引暂存器（Index register）、堆栈指针以及一般用途暂存器、状况代码等，主要用途在于中断时暂时存储数据，以便稍后继续利用；其数量及类因电脑架构有所差异。
- CPU排班法：优先级、排班队列等指针以及其他参数。
- 存储器管理：如标签页表等。
- 会计信息：如CPU与实际时间之使用数量、时限、账号、工作或进程号码。
- 输入输出状态：配置进程使用I/O设备，如磁带机。

在我们的程序中因为设计非常简单，所以PCB仅包括程序计数器，进程状态，CPU暂存器和进程编号。

进程切换的步骤

1. 保存处理机上下文，包括程序计数器和其他寄存器
2. 更新PCB信息
3. 把进程的PCB移入相应的队列，如就绪，在某事件阻塞等队列
4. 选择另一进程执行，并更新其PCB
5. 更新内存管理的数据结构
6. 恢复处理机上下文

save过程

Save是一个非常关键的过程，保护现场不能有丝毫差错，否则再次运行被中断的进程可能出错。

涉及到二种不同的栈：应用程序栈、进程表栈、内核栈。其中的进程表栈，只是我们为了保存和恢复进程的上下文寄存器值，而临时设置的一个伪局部栈，不是正常的程序栈

在时钟中断发生时，实模式下的CPU会将FLAGS、CS、IP先后压入当前被中断程序（进程）的堆栈中，接着跳转到（位于kernel内）时钟中断处理程序（Timer函数）执行。注意，此时并没有改变堆栈（的SS和SP），换句话说，我们内核里的中断处理函数，在刚开始时，使用的是被中断进程的堆栈

为了及时保护中断现场，必须在中断处理函数的最开始处，立即保存被中断程序的所有上下文寄存器中的当前值。不能先进行栈切换，再来保存寄存器。因为切换栈所需的若干指令，会破坏寄存器的当前值。这正是我们在中断处理函数的开始处，安排代码保存寄存器的内容

我们PCB中的16个寄存器值，内核一个专门的程序save，负责保护被中断的进程的现场，将这些寄存器的值转移至当前进程的PCB中。

Restart过程

用内核函数restart来恢复下一进程原来被中断时的上下文，并切换到下一进程运行。这里面最棘手的问题是SS的切换。

使用标准的中断返回指令IRET和原进程的栈，可以恢复（出栈）IP、CS和FLAGS，并返回到被中断的原进程执行，不需要进行栈切换。

如果使用我们的临时（对应于下一进程的）PCB栈，也可以用指令IRET完成进程切换，但是却无法进行栈切换。因为在执行IRET指令之后，执行权已经转到新进程，无法执行栈切换的内核代码；而如果在执行IRET指令之前执行栈切换（设置新进程的SS和SP的值），则IRET指令就无法正确执行，因为IRET必须使用PCB栈才能完成自己的任务。

解决办法有三个，一个是所有程序，包括内核和各个应用程序进程，都使用共同的栈。即它们共享一个（大栈段）SS，但是可以有各自不同区段的SP，可以做到互不干扰，也能够用IRET进行进程切换。第二种方法，是不使用IRET指令，而是改用RETF指令，但必须自己恢复FLAGS和SS。第三种方法，使用IRET指令，在用户进程的栈中保存IP、CS和FLAGS，但必须将IP、CS和FLAGS 放回用户进程栈中，这也是我们程序所采用的方案。

（三）、数据结构

cpuRegister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;
```

在cpuRegister中我们将寄存器的值保存在其中。每一个寄存器都用一个int来表示。

PCB结构设计

```
typedef struct{
    cpuRegister regImg;
    int pid;
    int status;
} PCB;
```

在PCB结构中，我们包含一个cpuRegister结构，还包含两个int型的变量，分别是`pid`和`status`，他们分别记录这个进程的编号和状态。我们用数值来表示进程的状态，0 -> new ；1-> ready ； 2-> running。

PCBList 结构设计

```
cpuRegister reg;
cpuRegister* preg = &reg;     //preg保存的是save里面的地址
PCB PCBList[10];
PCB* pn = PCBList;  //指向当前的进程控制块
```

我们定义了一个PCB类型的数组，数组包含10个数据单元，所以最多可以保存十个进程。另外我们还定义了一个PCB*的指针，来保存当前的进程控制块的地址，在最开始的时候初始化为PCBList的首地址。对于内核进程我们没有设置专门的进程控制块，而是定义了一个cpuRegister reg来保存现场，另外定义了一个cpuRegister*的指针来保存地址。这一设计其实主要是历史遗留问题，因为在实验五中我们保存内核的时候就是使用了这个变量，实验六在实验五的基础上进行更改所以这个变量就保留了下来。但事实上我认为这里其实将内核也保存在PCBList中是一个更好的设计。

（四）、代码讲解

Save

```
_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]
```

上述过程就是我们的save过程设计，这个过程是非常巧妙的，下面对于一些重点进行解释。

1. 在最开始先把ds段压栈保存，因为这个时候ds的内容可能是用户程序的，但是如果我们要访问内核定义的变量就要调整ds段寄存器，可是这个时候我们还没有保护现场，稍微执行一条指令都有可能导致现场被破坏，所以这个时候我们先压栈，这样后面只要pop就可以保证栈平

2. 将cs压栈后弹栈到ds中，这样就可以设置当前数据段和代码段一致。注意这个步骤必须放在在最开始，因为我们后面要访问的数据都是在内核数据段的，，只有先完成这个操作后面才能正确进行。
3. 因为陷入中断的时候，系统会自动将程序状态字，cs和ip自动压栈，所以我们要将他们pop出来才能保存。
4. 最后返回调用save的位置必须要用jmp而不能用ret，因为我们要将save和restart看成是和一个统一的过程。而且此时返回地址已经被我们弹栈，用ret是返回不了的。

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
```

和save过程相比，restart过程就要简单很多了，只需要将变量一个个的还原就可以了，但是在这里我们还是有一些细节要注意。

1. 一定要先把保存的栈还原然后再压栈PSW，cs和 ip！！！因为我们是在用户程序触发的中断，所以要在用户程序的栈压入PSW，cs和 ip。
2. 在其他变量保存结束之后记得要把ds和si恢复，注意这里用的是栈的方法，因为如果用变量，ds一旦改变就不能正确访问，而且si是作为索引的，如果si改变又不能正确索引到内存，所以这里一定要注意。
3. 中断返回之前的常规操作不能忘!

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

4. 这里最后的返回一定要用iret，这样才能和前面进入中断相对应，系统自动压入栈中的PSW，cs和 ip一定要利用iret再弹出才能保证栈的水平。

schedule

```
void schedule(){
    PCB* temp;
    temp = PCBList + CurrentPCBid;
    temp->status = 1;
    CurrentPCBid++;
    if(CurrentPCBid==ProgramCnt){
        CurrentPCBid = 0;
    }
    pn = PCBList + CurrentPCBid;
    PCBList[CurrentPCBid].status = 2;
}
```

我们的schedule过程在C语言里面实现，schedule的工作主要就是切换pn指针所指向的进程控制块单元，便于restart的时候恢复现场。同时切换进程的状态。

init_pcb

```
void init_pcb(PCB* pcb,int seg, int offset,int pid){
    pcb->regImg.ax = 0;
    pcb->regImg.bx = 0;
    pcb->regImg.cx = 0;
    pcb->regImg.dx = 0;
    pcb->regImg.si = 0;
    pcb->regImg.di = 0;
    pcb->regImg.cs = seg;
    pcb->regImg.ds = seg;
    pcb->regImg.es = seg;
    pcb->regImg.ss = seg;
    pcb->regImg.sp = 0xfff;
    pcb->regImg.bp = 0;
    pcb->regImg.ip = offset;
    pcb->regImg.flag = 512;
    pcb->status = 0; //设置成new
    pcb->pid = pid;
}
```

在这个过程中我们初始化PCBList的单元，具体做法就是将pcb单元中的寄存器变量的内容根据传入的参数进行初始化，过程非常简单这里就不再赘述。

_initest

```
_initest:
```

```
push es
push si
push di
push ax
push bx
push cx
push dx
push bp
push ds


mov cl,12
mov ax,2000h
mov es,ax
mov bx,0100h
mov ax,0201h
mov dx,0000h
mov ch,00h
int 13h

mov cl,13
mov ax,6000h
mov es,ax
mov bx,0100h
mov ax,0201h
mov dx,0000h
mov ch,00h
int 13h

mov cl,14
mov ax,4000h
mov es,ax
mov bx,0100h
mov ax,0201h
mov dx,0000h
mov ch,00h
int 13h

mov cl,15
mov ax,5000h
mov es,ax
mov bx,0100h
mov ax,0201h
mov dx,0000h
mov ch,00h
int 13h

pop ds
pop bp
pop dx
pop cx
pop bx
pop ax
pop di
pop si
```

```
    pop es
    mov word[ProgramCnt],4
    ret
```

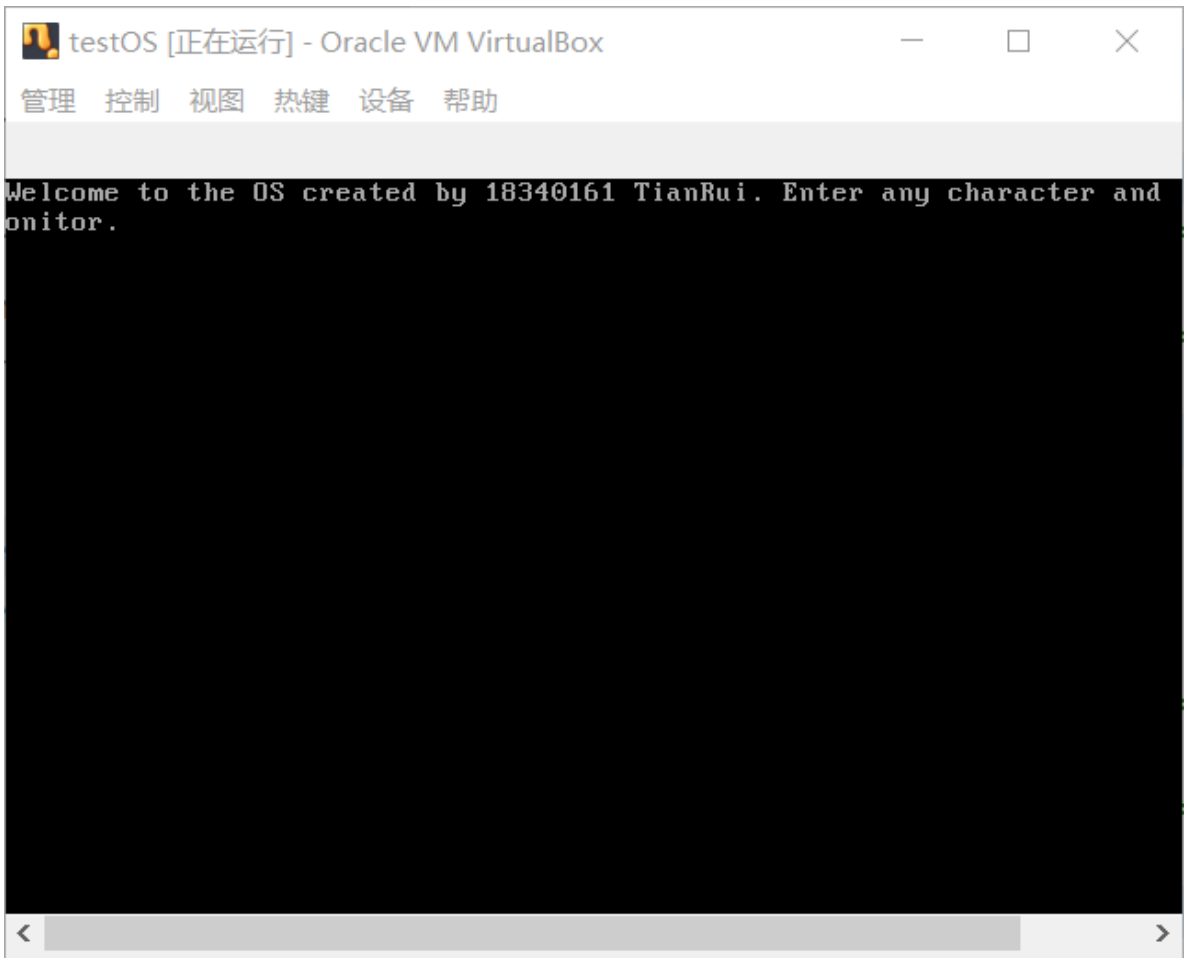
我们这里还定义了一个_initest过程来加载程序到内存，同时很重要的一个点就是要将变量ProgramCnt的内容更改为我们程序的数量，这里调用四个程序就是4，因为我们后面是要根据ProgramCnt的值进行有无变量跳转的，所以这里一定要记住更改。

int22h

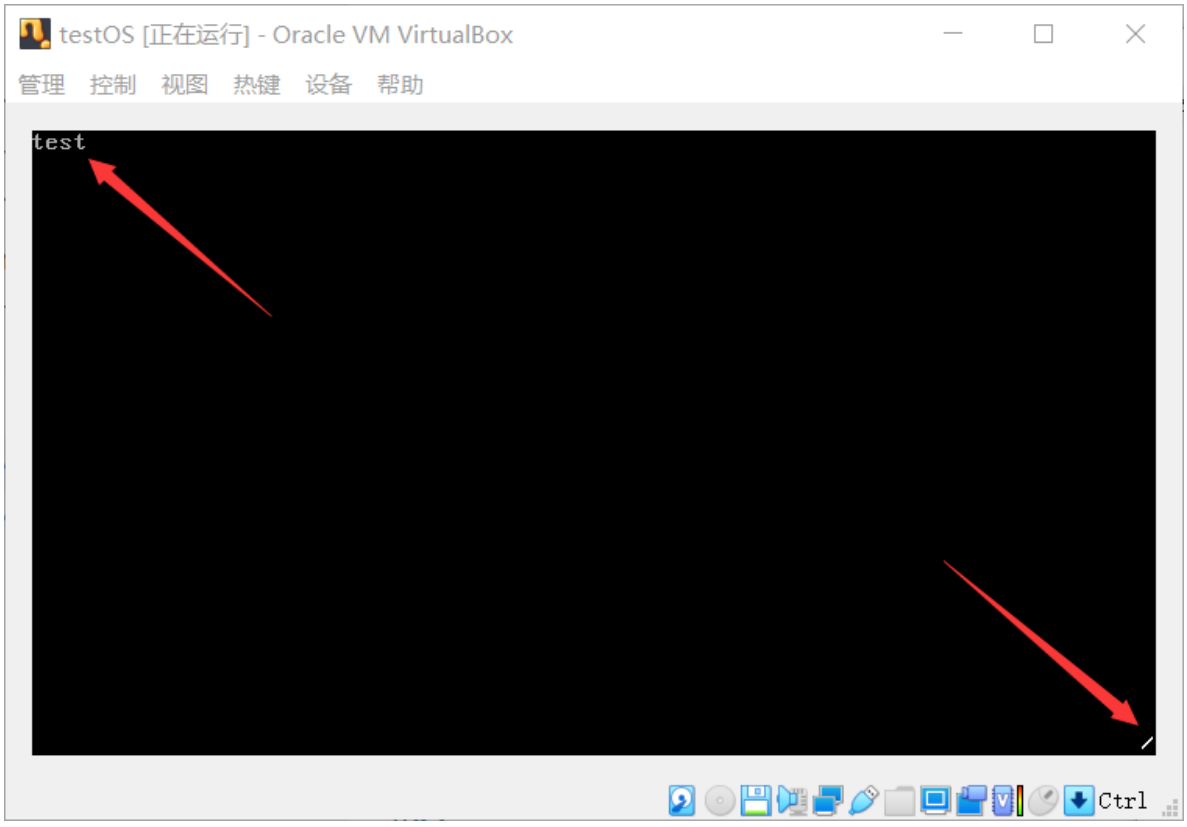
```
int22hsys:
    cli

    push ax
    mov ax, reg
    mov word[preg],ax
    pop ax

    call _save
    push bx
    push ds
    mov bx,cs
    mov ds,bx
    mov bx, pn
    mov bx,[ds:bx]
    mov word[preg], bx
    pop ds
    pop bx
    jmp _restart

end22hsys:
    jmp _restart
```

这里我们将实验五中没有用上的系统调用进行了一个修改，让int22h中断触发整个并行程序的启动，这一设计是有意义的，因为更改完这些之后我们需要一个流程来启动所有的程序，但是如果是非中断过程调用save和restart会导致栈的内容不对，即使是自己压入psw和cs也不能保证栈的水平，所以我们对22h中断进行了改造，来启动我们的并发程序。

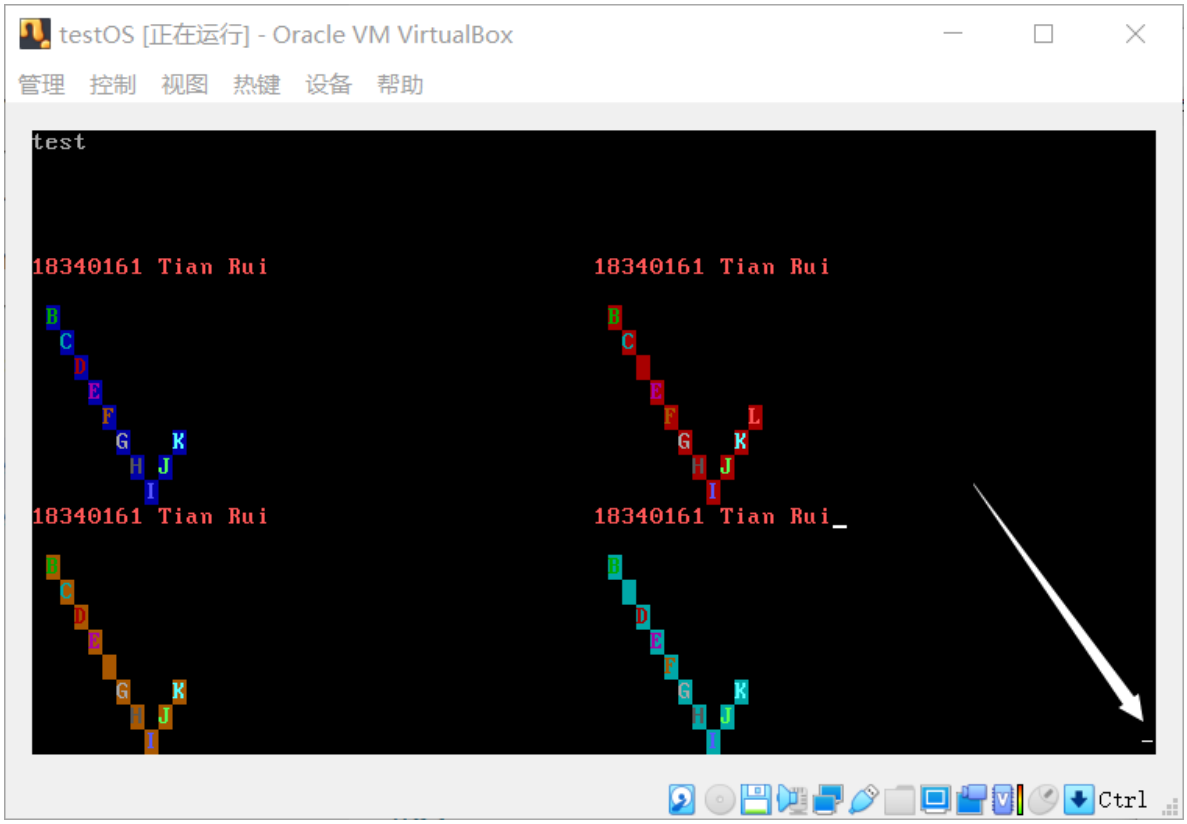
六、实验结果



首先是引导扇区，按下任意键后进入内核程序

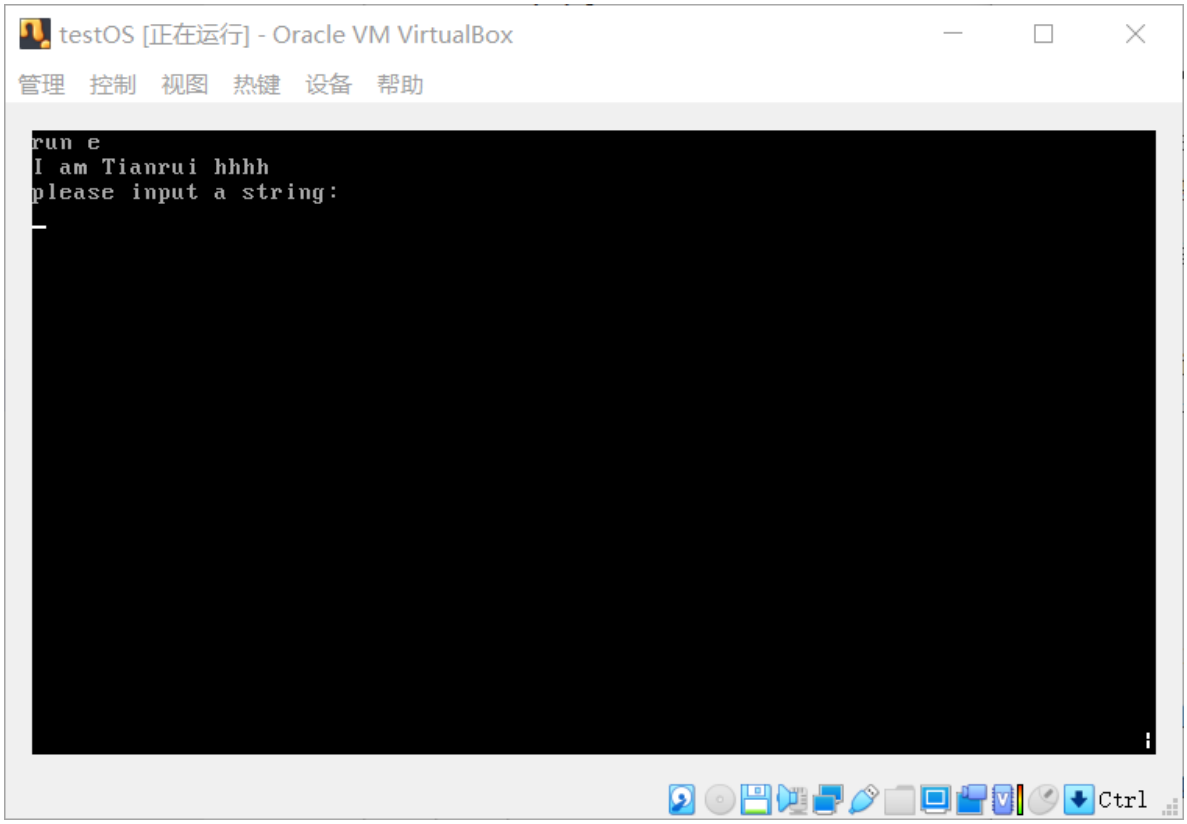


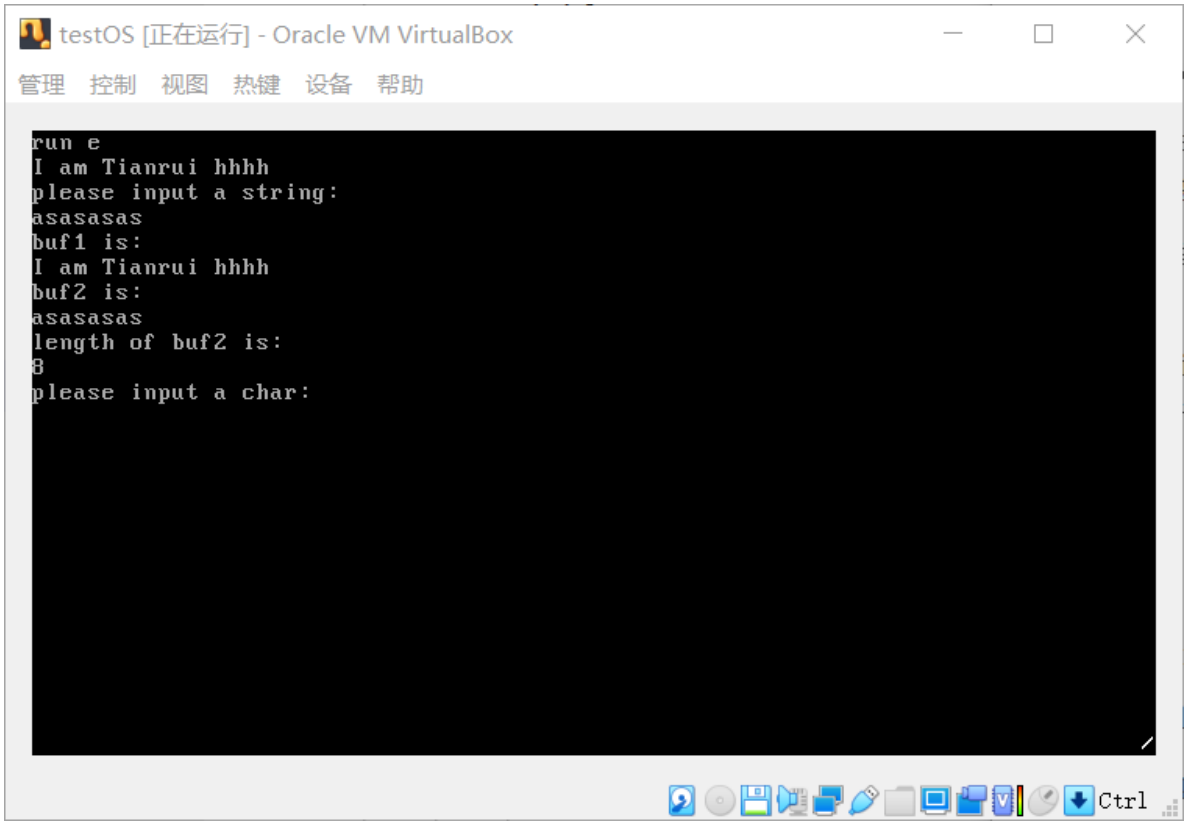
输入指令`test`,可以看到这个过程风火轮也是一直在转的。

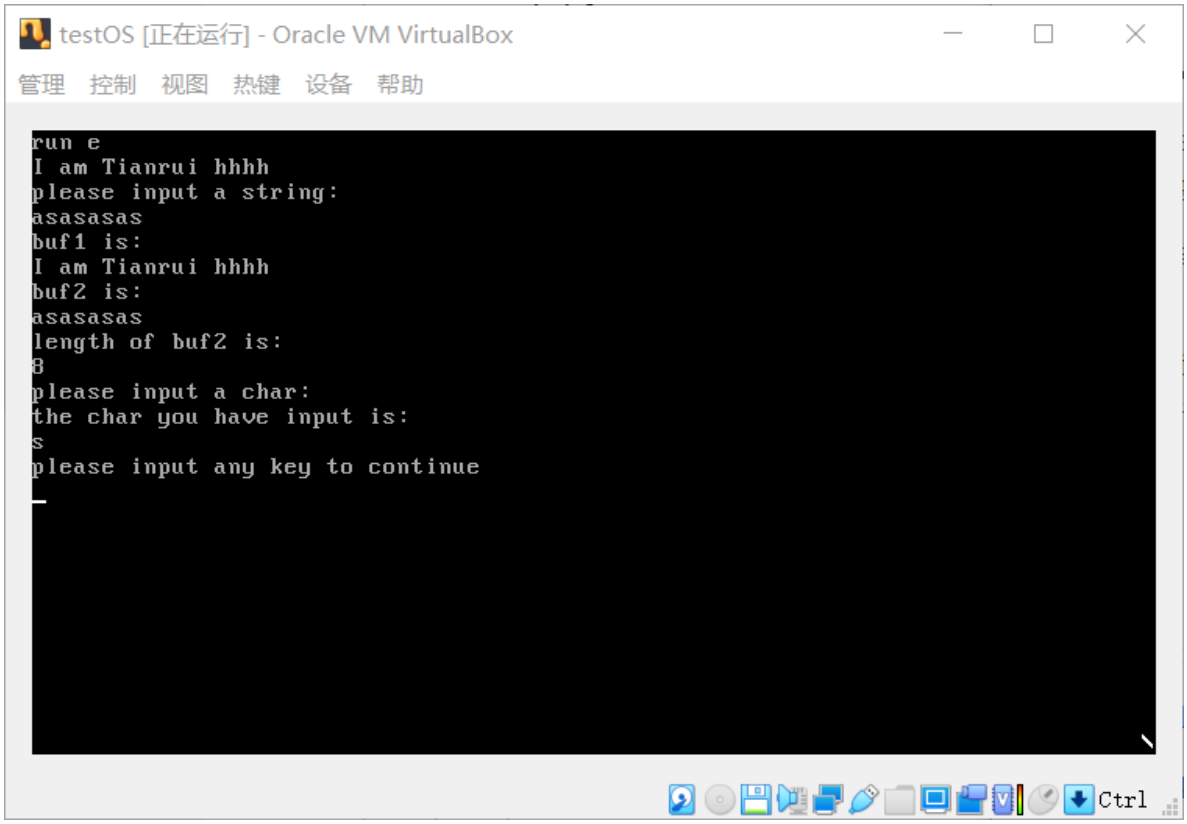


四个程序同时运行，风火轮仍然在转

同时实验五中实现的过程除了对int22h进行了改造之外，其余功能仍然保留。下面只展示简单的功能。


testOS [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
run e
I am Tianrui hhhh
please input a string:
Ctrl


testOS [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
run e
I am Tianrui hhhh
please input a string:
asasasas
buf1 is:
I am Tianrui hhhh
buf2 is:
asasasas
length of buf2 is:
8
please input a char:
Ctrl



七、实验心得

这次实验是几个实验写的最快的一个了，一个是因为之前已经完成了save和restart过程，所以在这个基础上进行修改就简单了很多，也还因为随着课程的深入，对操作系统的理解也更加深刻，所以进行试验也越来越简单。虽然这次完成的很快，但是还是遇到了很多苦难，下面我们一一讲解。

1. 开始的时候将变量ProgramCnt和CurrentPCBid定义在了汇编部分，这样程序在计算偏移量的时候总是会内存访问错，原来是因为我只给这两个变量申请了一个字的内存，但是在C中进行计算的时候却会将四个字节的内容进行相加，这样的话如果这个位置的内容最开始不是0，那么前面两个字节的内容就会被当作这个变量的空间参与计算，解决方法很多，我们这里采用将变量挪到C语言中定义成全局变量再extern到汇编语言中，问题解决~
2. 在我们的程序设计中，如果当前ProgramCnt内容是0，就说明不需要schedule过程，就会直接执行风火轮过程。所以就想着这一部分能不能不save了呢，开始我是这样操作的，而且为了方便debug，就把main函数中的gets函数来获取指令的语句注释掉，换成了直接将指令缓冲区直接定义成我们要测试的指令，所以程序没有问题可以正确运行，可把我高兴坏了，最后就想很开心的把gets函数来获取指令的语句换回来的时候发现，输入不了orz。这个真的难受了，本来以为都可以了，只能重新debug。因为输入是涉及到系统中断的，所以这个过程很难debug，开始一直单步调试了很久都改不对。考虑到在之前的实验中一直都可以输入，所以我果断排除了是gets函数出了问题的可能性。和实验五中相比，较大的改动就只有Timer过程，所以我着重注意Timer的变化，我一点一点注释掉在实验六中新加入的语句，看注释到哪一段的时候程序变得能正确执行了，然而我发现我基本上全部都注释掉了，只剩下风火轮的时候还是不能正确执行，这个时候我醍醐灌顶幡然醒悟虎躯一震！会不会是因为没有save所以不能正确执行了！！！这时我先将实验五的save也去掉，发现实验五也不能输入了，然后将实验六的Timer进行调整，无论当前ProgramCnt内容是几，都要进行save过程，果然程序可以正确执行啦~
3. 正当我开心的不得了的时候，我发现程序又不能正确返回了orz。不过没有关系，小小的困难不能将我击倒！打开bochs继续debug！这个时候发现是跳入的时候用了简单粗暴的jmp跳转到我们要运行的程序，这样程序运行结束restart的时候是没有内容

> 可以让他跳转回内核的，所以一定要将内核的现场也保存，这样才能让程序正确的再跳转回内核程序，所以我开始在_initest过程中直接调用了save和restart。但是这样做是不对的，因为save和restart是针对中断设计的，如果在普通的过程中直接调用，栈会出问题的。因为是栈的问题，所以我最开始决定手动压栈程序状态字和cs，但是还是一直出错，因为少了一个ip，这个ip如果我们自己计算的话非常不现实，所以改变策略，将之前没有用到的int22h进行了修改，让这个中断来实现程序的启动，果然更改之后程序正常运行！！！

总体来说这次实验完成的是最快的一次也是收获最大的一次，甚至在这次实验过程中收获到了OS的乐趣。能够实现多进程的并发是一件很有成就感的事情，希望以后可以再接再厉设计出更加完善的OS。

八、参考文献

[1]刘斯宇.二状态进程模型[R].广东省广州市:SYSU University,2019.
[2]百度百科.进程控制块[EB/OL].https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E6%8E%A7%E5%88%B6%E5%9D%97/7205297?fr=aladdin#2,2018-08-10.
[3]王道论坛.操作系统考研复习指导[M].电子工业出版社:北京,2020:33-.