实验六：实现时间片轮转的二态进程模型

18340225 钟婕

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

实验要求

1、了解操作系统内核的二态进程模型

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。

4、修改时钟中断处理程序，调用时间片轮转调度算法。

5、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

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。

6、实验5的内核其他功能，如果不必要，可暂时取消服务。

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

  本次实验是在实验五的基础上，对于实验五的中断处理程序的保护现场Save过程以及恢复现场Restart过程做出适当修改，且对于时钟中断的中断处理程序进行修改，使得利用时钟中断触发，调用Save和Restart过程实现多个进程的分时调度过程。通俗而言，本次实验的核心就是在时钟中断处理程序中调用Save过程：保存当前被中断程序的所有寄存器上下文并置被中断进程为就绪态，接着再利用内核程序的进程调度过程，切换调度下一个处于就绪态的进程为运行态，并调用Restart过程，恢复被调度进程的所有寄存器上下文值，恢复到被调度程序被中断处开始执行。

  ## 实验原理

  根据实验内容和要求可知，本次实验是在实验五的基础上，修改实验五编写的简易的中断处理程序中的保护现场Save过程以及恢复现场过程Restart过程，并且将此保护现场Save过程以及恢复现场的Restart过程融合在时间中断的中断处理程序中，充分利用时间中断的触发来进行多个进程的调度，实现多进程的分时执行。除此之外，为了实现多进程的分时执行，还需要在实验五的基础上进一步设计并完善进程结构相关的数据结构，并利用此数据结构实现进程调度。为了顺利完成实验，需要了解有关进程状态、进程模型、进程表以及进程交替执行等相关原理，现将原理叙述如下：

  1. **二状态的进程模型**

     - **进程模型就是实现多道程序和分时系统的一个理想的方案。**

       多个用户程序并发执行

       进程模型中，操作系统可以知道有几个用户程序在内存运行，每个用户程序执行的代码和数据放在什么位置，入口位置和当前执行的指令位置，哪个用户程序可执行或不可执行，各个程序运行期间使用的计算机资源情况等等。

     - **二状态进程模型**

       执行和等待

       目前进程的用户程序都是COM格式的，是最简单的可执行程序

       进程仅涉及一个内存区、CPU、显示屏这几种资源，所以进程模型很简单，只要描述这几个资源。

     - **以后扩展进程模型解决键盘输入、进程通信、多进程、文件操作**

2. **初级进程**

- 现在的用户程序都很小，只要简单地将内存划分为多个小区，每个用户程序占用其中一个区，就相当于每个用户拥有独立的内存
- 根据我们的硬件环境，CPU可访问1M内存，我们规定MYOS加载在第一个64K中，用户程序从第二个64K内存开始分配，每个进程64K，作为示范，我们实现的MYOS进程模型只有两个用户程序，大家可以简单地扩展，让MYOS中容纳更多的进程
- 对于键盘，我们先放后解决，即规定用户程序没有键盘输入要求，我们将在后继的关于终端的实验中解决
- 对于显示器，我们可以参考内存划分的方法，将25行80列的显示区划分为多个区域，在进程运行后，操作系统的显示信息是很少的我们就将显示区分为4个区域，用户程序如果要显示信息，规定在其中一个区域显示。当然，理想的解决方案是用户程序分别拥有一个独立的显示器，这个方案会在关于终端的实验中提供
- 文件资源和其它系统软资源，则会通过扩展进程模型的数据结构来实现，相关内容将安排在文件系统实验和其它一些相关实验中

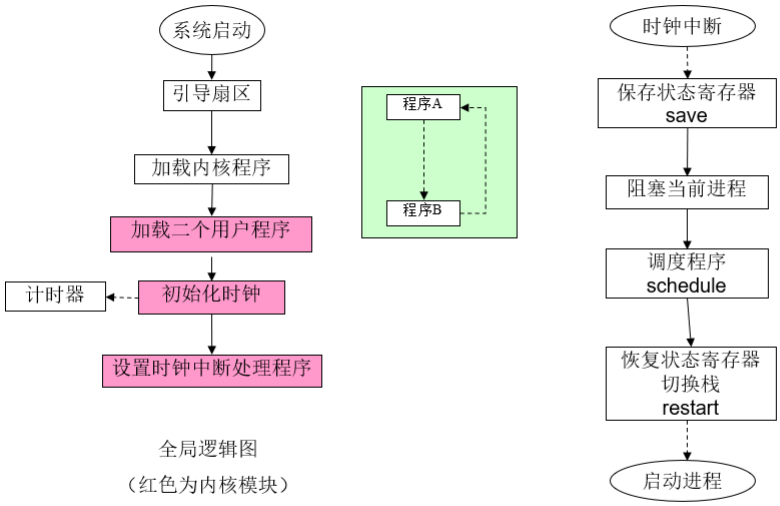
3. **进程表**

初级的进程模型可以理解为将一个CPU模拟为多个逻辑独立的CPU。每个进程具有一个独立的逻辑CPU。同一计算机内并发执行多个不同的用户程序，MYOS要保证独立的用户程序之间不会互相干扰。为此，内核中建立一个重要的数据结构：进程表和进程控制块PCB。现在的PCB它包括进程标识和逻辑CPU模拟。逻辑CPU包含8086CPU的所有寄存器：AX/BX/CX/DX/BP/SP/DI/SI/CS/DS/ES/SS/IP/FLAG，这些寄存器用内存单元模拟。逻辑CPU轮流映射到物理CPU，实现多道程序的并发执行。我们可以在汇编语言中描述PCB亦可以用C语言描述PCB。

4. **进程交替执行原理**

- 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
- 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

  我们可以简要利用下图理解实现进程模型的系统框架：



全局逻辑图

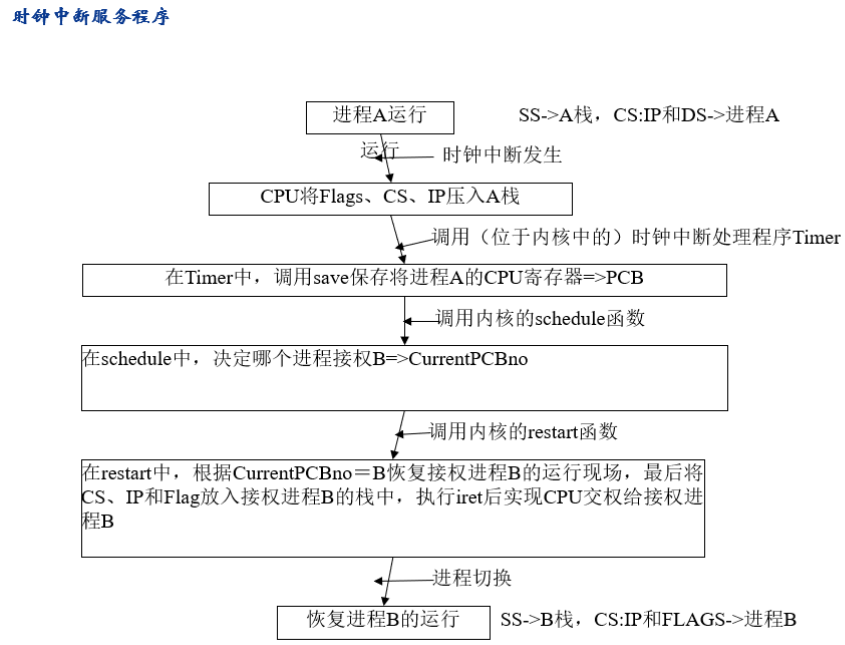
（红色为内核模块）

5. **内核**

- 利用时钟中断实现用户程序轮流执行

- 在系统启动时，将加载两个用户程序A和B，并建立相应的PCB。
- 修改时钟中断服务程序：每次发生时钟中断，中断服务程序就让A换B或B换A。要知道中断发生时谁在执行，还要把被中断的用户程序的CPU寄存器信息保存到对应的PCB中，以后才能恢复到CPU中保证程序继续正确执行。中断返回时，CPU控制权交给另一个用户程序。

  下图可以简要显示利用时钟中断处理程序实现进程调度的流程：



6. **保护现场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。

7. **恢复现场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。

8. **进程调度: Scheduler过程**

   采用时间片轮转法，每次运行时间到了(即触发时钟中断)就会调用Scheduler方法来切换到下一个要运行的用户程序或者内核程序。

- **PCB相关设计(含多进程调度函数)**

从实验五设计保护现场Save过程以及恢复现场Restart过程和本次实验的实验原理、要求可知，中断处理过程中的保护现场以及恢复现场过程，主要是保护被中断进程的所有上下文寄存器，恢复下一被调度进程的所有上下文寄存器等必要信息。而对于进程的这些必要信息的保存，我们不能仅仅依靠一些基本类型变量实现，因此我们的目标是要建立一个进程的物理表示—寄存器的进程映像，这样的一个数据结构可以让我们在进行多个进程的调度时，将上一个进程的所有上下文寄存器等必要信息保存该进程对应的PCB结构中，而对于下一个被调度进程的PCB结构中的各个数据成员可以将之顺利恢复给各个对应的寄存器。因此，我们需要定义一个PCB相关的结构体来模拟进程映像。

  - **PCB结构设计**

    对于简易进程映像的设计，首先需要考虑到寄存器相关的结构。由于在Save过程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，设计一个寄存器的进程映像结构，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，结合本次实验需要实现多进程的调度执行可知，要想实现多进程的调度执行，我们就需要区分各个进程的状态，便于内核对于各个进程的调度，本次实验的多进程调度要求仅需几个基本状态即可区分各个进程的执行过程，基本的二状态进程模型：执行态与等待态即可满足。考虑到在理论课学习的过程，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，因此除了上述的二状态基本模型包含的两个状态外，我还在PCB进程控制块中加入了进程状态中的就绪态、运行态、新建态这三种状态构成一个进程状态数据成员，用以保存当前被中断程序的状态。

上述的两大部分：寄存器的进程映像与进程状态是构成我的PCB进程控制块的两大基本部分。当然，结合理论课的学习可知，对于PCB进程控制块的定义中还包含进程标识、程序名等信息，因此加上这些标识就可以构成整个PCB进程控制块。

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
    int  ID;//进程号
    char * name;//程序名
}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，且内核程序和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[1~4]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，同时还对于进程的各个状态:NEW、Ready、Run、Exit设定了对应的值。

具体的数据设计如下：

```
int NEW = 0;
int Ready = 1;
int Run = 2;
int Exit = 3;
PCB pcb[10];
int cur_pnum = 0;/*0下标为内核程序，1~4为四个用户程序*/
int process_num = 0;
```

- **PCB结构相关函数设计**

在设计了PCB进程控制块的简易结构之后，需要设计相关的函数便于之后时间中断处理程序的Save和Restart过程的使用。

根据中断处理的保护现场、恢复现场以及多进程分时运行调度的思想，我们很自然地联想到需要设计以下几个功能的函数：

1. 初始化内核程序以及四个用户程序的PCB结构函数；
2. 将被中断进程的上下文寄存器值保存在被中断进程对于的PCB结构中的函数；
3. 将被调度进程的对应的PCB结构返回给Restart过程的函数；
4. 多进程分时运行所需的调度函数；
5. 首次运行的进程状态更新函数；

下面分别介绍这几个函数。

- **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
    p->Process_Status = NEW;//进程状态
}
```

特别说明，在本次实验中不同于实验五，内核程序的进程状态可以初始化为NEW，因为本次实验中的时钟中断处理程序中的Save和Restart过程不是针对同一个进程的，即Save的是上一个被中断的进程，而Restart的是下一个被调度的进程，而且本次实验设计二状态进程模型，所以虽然内核程序不是第一次执行，但由于在进程调度时会将内核置为就绪态并调度下一进程，因此不会受到初始化为NEW状态的影响。

- **保存上下文寄存器值的函数**

在多进程的分时运行中，由于是由时钟中断触发引起的，因此在时钟中断处理程序的Save过程中，我们需要将上一个被中断进程的所有必要信息，含：所有上下文寄存器的值，保存在该被中断进程对应的PCB结构中。这样才便于下一次调度该程序时完整地恢复现场。

显然，对于Save过程中的保护被中断进程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。特别地，由于被中断程序的ip、cs、psw是由调用时钟中断时自动入栈的，因此在汇编的Save过程中调用Save_PCB函数时，ip、cs、psw三个值不需要再入栈。

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;

    pcb[cur_pnum].RI.IP = ip;
    pcb[cur_pnum].RI.CS = cs;
    pcb[cur_pnum].RI.FLAGS = flags;
}
```

- **多进程分时运行的调度函数**

  根据实验要求，需要在时钟中断处理程序中利用Save、Restart过程实现多进程的切换执行，因此在Save过程保存了上一个被中断进程的所有必要信息之后，需要利用一个进程调度的Process_Schedule函数，调度下一个将被执行的进程。

  整个调度过程包含：首先需要将上一个被中断进程的进程状态置为就绪态，等待下一次被调度；接着需要将变量cur_pnum(表示当前被执行进程的进程控制块下标)递增(因为根据实验要求，本次实验的多进程调度是顺序的，因此加一即可)，当然本次实验设计的可以同时执行的多个程序的命令，由于同时执行的进程数是一定的，因此cur_pnum变量的递增需要注意范围；最后,我们需要将更新后的cur_pnum值对应的进程的进程状态置为执行态。

  具体代码如下：

```
void Process_Schedule()
{
    pcb[cur_pnum].Process_Status = Ready;
    cur_pnum++;
    if (cur_pnum > process_num)
        cur_pnum = 1;/*多个用户程序交替执行直到都执行完成，因此从最后一个程序调度第一个程序需要置1*/
    if (pcb[cur_pnum].Process_Status != NEW)
        pcb[cur_pnum].Process_Status = Run;/*特别注意！！！！坑！！第一次运行的程序和已经运行的程序再执行是不同的！*/

}
```

- **首次运行的进程的进程状态的更新函数**

  从实验五的Restart的测试过程可知，对于首次运行的进程(即进程状态对应为NEW的进程)，在首次被调度时，只需要利用Restart过程将PCB结构中的各个数据成员的值重新赋值给寄存器即可；而对于非首次运行的进程(即进程状态对于为Ready的进程)，由于不是第一次被调度，因此在之前曾经历过Save过程，由于在Save过程是利用栈操作的，而栈寄存器ss入栈前，ss的值被改变了，因此需要单独处理。因此当发现被调度进程的进程状态为为NEW时，需要将之更新为Ready，这样在该进程下一次被调度时即满足了非第一次执行的条件，才能顺利辨别。

  具体函数代码如下：

```
void Fornew()
{
    if (pcb[cur_pnum].Process_Status == NEW)
        pcb[cur_pnum].Process_Status = Run;
}
```

- **获取被调度进程对应的PCB结构的函数**

由于在Restart过程中，需要将下一个被调度进程的PCB结构中的各个寄存器值重新赋值给对应的各个寄存器，因此我们需要设计一个函数返回在pcb数组中对应于当前被调度程序的PCB结构。

函数设计相对简单，只需要根据cur_pnum(即当前被调度程序对应的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过程和Restart过程设计**

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。虽然在实验五中，我们进一步设计了中断处理过程中的保护现场Save过程与Restart过程，但是在实验五中的保护现场的Save过程与恢复现场的Restart过程的设计还是相对简单，我们只考虑对于单个进程触发中断时的保护现场(即保护当前被中断程序的所有寄存器上下文)，而在执行完中断处理程序后的恢复现场，我们也只考虑了对于当前被中断程序的恢复，而不涉及多个进程的调度。而在本次实验中，我们所需的多进程分时运行要求，Save过程要保存上一个被中断进程的所有必要信息，Restart过程需要将下一个被调度进程的所有必要信息予以恢复。

下面将分别具体叙述Save过程、Restart过程在本实验的设计。

  - **Save过程设计**

根据上述的PCB数据结构设计以及PCB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。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，会将被中断程序中断的ip、cs、PSW均入栈，因此在调用Save_PCB函数过程中我们不需要再将ip、cs、PSW入栈，只需要将其他寄存器的值根据Save_PCB函数的参数顺序设计，依次入栈即可。

首先，我们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状态。

除此之外，由于本次实验是以多进程的分时调度执行为中心的，因此不同于实验五，我们在调用Save_PCB函数保存上一个被中断进程的所有上下文寄存器值之后，还需要利用我们在PCB相关函数设计模块叙述的**多进程调度Process_Schedule**函数，进行进程的调度，确定下一个被调度的进程，为Restart过程：执行下一被调度进程做准备。关于Process_Schedule函数的具体设计已经在PCB函数设计模块详述了，在此不再赘述。

部分代码如下所示：

```
Save:
    push ss
    push ax
    push bx
    push cx
    push dx
```

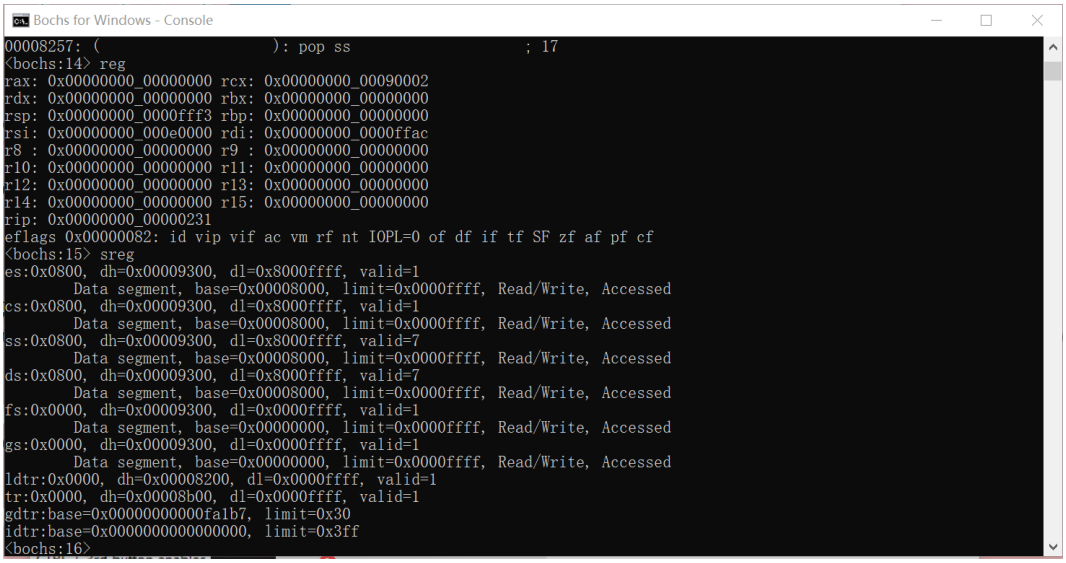
```
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    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
    call near ptr _Process_Schedule;利用多进程调度函数，调度下一被执行进程
    ;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
```

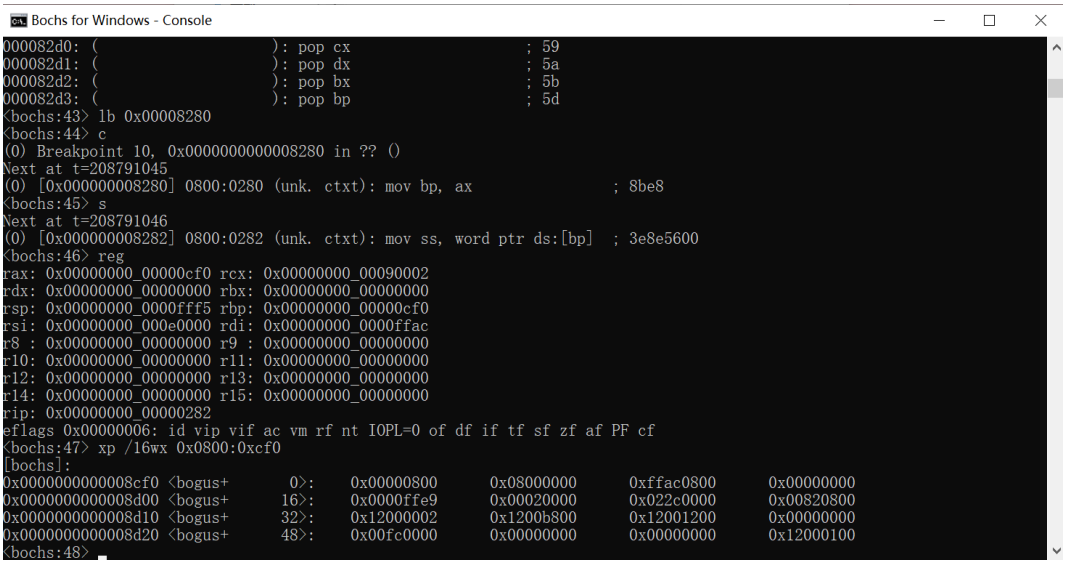
特别注意地是，由于涉及到段内过程调用，因此在call指令调用函数时，一定要记得重新给段寄存器赋值，以防万一。在用Bochs单步调试测试过程中，最初发现用户程序触发时钟中断时会出现一直无法成功调用Save_PCB函数的情况，后来经过Bochs单步调试以及利用sreg命令查看cpu寄存器的值之后，才知道原来是由于用户程序的段地址与内核程序的段地址是不同的，当用户程序返回时，CS寄存器的值会更新成内核的段地址，但此时ES、DS寄存器的值还是保留着原来用户程序的段地址，因此会出现错误。因此在调用之前需要利用已更新的CS寄存器的值重新给ES、DS寄存器赋值，这样才能确保调用过程的顺利执行。

为了确保上述在实验五基础上修改的Save过程能够正确执行，利用Bochs进行验证：

首先利用sreg命令查看被中断进程在被中断时刻的各个寄存器的值，具体如下：



接着我们在Save过程中根据Save_PCB函数参数情况将参数依次入栈，并调用Save_PCB函数将被中断进程的各个寄存器的值保存在相应的PCB结构中，利用xp命令查看对应PCB结构的值如下：



通过对比两张图片所示的各个寄存器数据值可知，上述的Save过程可以成功、正确地将被中断进程的所有上下文寄存器等信息成功保存至被中断进程相应的PCB数据结构中。

- **Restart过程设计(含多进程调度函数)**

  根据本次实验的实验要求可知，在多进程的调度执行中，Restart过程主要是将下一个被调度进程的PCB数据结构中的各个数据成员重新恢复给各个寄存器，并进行中断返回。

  根据上述Save过程设计的思想，Save过程在调用完Save_PCB函数之后，调用了**多进程调度函数Process_Schedule**进行多进程分时执行的调度，因此在Restart函数，我们其实就是将在Save过程中被调度的下一进程保存在对应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，因此需要将被中断程序的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。特别地是，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，对于实验五中出现的当进程不是第一次执行时，进程对应的PCB结构中的sp寄存器的值不能成功访问栈的问题同样存在，因此我们同样需要类似于实验五，通过对当前被调度进程的进程状态是否为NEW进行判断，若不为NEW，则当前被调度进程不是第一次执行，需要将sp的值加上18，否则不用。这样进程才能成功重启执行。

另外由于，本次实验是利用时钟中断实现多进程的调度操作的，因此Save和Restart过程是描述在时钟中断处理程序中的，根据实验要求我们需要保存无敌风火轮显示，因此在Restart过程中，我们需要调用之前几个实验撰写的无敌风火轮程序段，再进行获取当前被调度进程PCB结构，恢复各个寄存器值的操作。

首先展示调用原无敌风火轮程序段、获取当前被调度进程PCB结构以及处理进程SP寄存器值的过程的相关代码：

```
redone:
    mov ax,cs
    mov es,ax
    mov ds,ax
    call near ptr Timer;照样需要显示无敌风火轮

    mov ax,cs
    mov es,ax
    mov ds,ax

    call near ptr _Current_PCB;获取当前被调度进程的PCB结构
    mov bp,ax;将返回值赋予bp寄存器

    mov ss, word ptr ds:[bp];恢复ss
    mov sp, word ptr ds:[bp+16];恢复sp

    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
```

```
Not_First_Time:
     add sp,16 ;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 Restart
```

接下来展示，如何将获取的被调度进程的PCB结构中的各个数据成员重新赋值给各个寄存器。

```
Restart:
    call near ptr _Fornew

    ; 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
```

上述代码就是按照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。

- **时钟中断处理程序设计**

根据实验要求，我们需要利用时钟中断处理程序完成多进程的分时执行，因此我们需要修改原来的时钟中断处理程序。由于本次实验中是要求利用时钟中断的定时触发来实现对于多进程的调度执行，因此对于时钟中断处理程序的设计需要考虑以下几个方面：首先我们需要设定时钟中断的触发频率，以此让我们的多进程能够平稳执行；其次，我们需要修改时钟中断的中断向量表，这样在触发时钟中断时才能够顺利进入我设计的中断处理程序中执行；接着，由于时钟中断是多进程切换的契机，因此我们需要将上述的Save和Restart过程融合在中断处理程序中；最后我们需要考虑中断处理程序的结束操作。

下面我将分别叙述时钟中断处理程序的多个设计方面：

- **设定时钟中断频率**

  根据实验要求，我们需要利用时钟中断的触发来进行多进程的调度，因此时钟中断的频率就是多进程切换执行的频率。为了能够让多进程的切换执行保持平稳，既不太快也不会太慢，因此我们需要利用34h控制字向43h的控制端口写入控制命令，并将自己设定的时钟中断频率(即计数器的值)发往40h所指的计数器0，以此设置时钟中断的每秒中断次数。

  具体代码如下所示：

```
;通过往端口 0x43 写入控制字来设置工作方式。往端口 0x40 写入计数初值。
public _SetTimer
_SetTimer proc
    push ax
    mov al,34h   ; 设置控制字
    out 43h,al   ; 利用out指令，将写控制字写到控制字寄存器
    mov ax,29830 ; 设置每秒 20 次中断（50ms 一次），控制中断的速度，ax即为计数器
的值
    out 40h,al   ; 将ax计数器低位的值写入计数器 0 的低字节
    mov al,ah    ; AL=AH
    out 40h,al   ; 将ax计数器高位的值写入计数器 0 的高字节
    pop ax
    ret
_SetTimer endp
```

- **修改时钟中断向量**

从实验四的中断向量的设置可知，对于中断处理程序的编写的一个重要环节是修改中断向量表，我们直到在内存区域的0~1023地址处存放着32个中断的中断向量，每个中断向量占32bits，每个中断对应的中断向量起始位置是中断号×4，存放着中断处理程序入口的段地址和偏移地址，因此类似于实验四，我们即可修改中断表：

```
;此时需要分时并行用户程序，因此需要切换时钟中断处理程序
public _ChangeTimer
_ChangeTimer proc
    push ax
    push es

    ;设置时钟中断频率，原来是18.206Hz,改成每秒20次
    call near ptr _SetTimer
    xor ax,ax
    mov es,ax
    mov word ptr es:[20h],offset TimerPro ;改时间中断向量
    mov word ptr es:[22h],cs

    pop ax
    mov es,ax
    pop ax
    ret
_ChangeTimer endp
```

- **多进程调度的总次数设置**

由于本次对于四个用户程序的执行，是多个程序作为若干个进程由内核程序分时调度执行的，因此若干个用户程序执行结束时即需要回到内核状态，此时的时钟中断处理程序不需要调度多进程，而只需显示原来的无敌风火轮即可。因此，对于多进程执行结束切换回内核模式的操作需要我们设定一个总的调度次数实现，设置一个变量Sche_Num记录调度次数，当变量值到达总的调度次数上限时，即结束多用户进程的执行，回到内核状态，此时中断处理程序只需显示无敌风火轮即可。

那么如何判断此时程序处于内核状态?我们可以利用之前设计的process_num变量，当有多用户进程执行时，process_num值为用户进程的数量，而当结束多用户进程执行时(即调度次数到达总调度次数上限时)，我们需要将process_num值设置为0，以此判别程序是否处于内核状态，若处于则此时时钟中断处理程序只需显示风火轮即可。

下面展示判别内核状态并显示无敌风火轮的代码：

```
    cmp word ptr[_process_num],0;判别是否内核状态
    jnz PreSave
    jmp None_Program

None_Program:
    call near ptr Timer;调用原时钟中断风火轮显示程序段
    push ax
    mov al,20h
    out 20h,al          ;发送EOI到主8529A
    out 0A0h,al           ;发送给从8529A
    pop ax
    iret
```

上述代码中，我将原来的无敌风火轮显示的代码封装成一个程序段Timer供新编的时钟中断处理程序调用，另外别忘了时钟中断处理程序结束处需要发送EOI信号给主从8259A。

而当process_num不等于0时，我们需要进行多用户进程的切换执行，如上述的总调度次数设定，当Sche_num变量值达到设定的总调度次数上限时，我们需要重置Sche_num变量并置process_num为0，以此告知程序：当前已结束多用户进程的执行，回到内核状态。而对于其他一些在PCB相关函数中使用的变量，如cur_pnum等，也需要设置成初始化的值。

下面展示具体代码：

```
;设定一个计数器记录分时并行是的总的调度次数
Sche_num dw 0
TimerPro:
    cmp word ptr[_process_num],0
    jnz PreSave
    jmp None_Program
PreSave:
    inc word ptr [Sche_num]
    cmp word ptr[Sche_num],500
    jnz Save
    ;达到总调度次数500
    mov word ptr[_process_num],0;告知程序处于内核状态
    ;变量重置初始化
    mov word ptr[_cur_pnum],0
    mov word ptr[Sche_num],0
    mov word ptr[_segment],2000H
    jmp redone
```

- **Save和Restart过程的融合**

在之前的Save和Restart过程设计部分，我已经详述了这两部分程序段的设计，因此在本次实验编写的时钟中断处理程序的主体部分我们只需要将Save和Restart部分融入进新编写的时钟中断处理程序TimerPro中即可。

在此，可以大概总结一下Save和Restart的操作思路：Save将上一个被中断进程的所以必要信息保存至被中断进程对应得PCB结构中，接着Process_Schedule调度函数调度下一个进程执行，并先调用原无敌风火轮显示程序段Timer显示无敌风火轮，再获取下一个被调度的进程的PCB结构，Restart过程将各个数据成员的值重新赋值给各个寄存器，完成多用户进程的切换执行。

下面简要展示两部分的串接结构：

```
TimerPro:
    ...
    Save
    call near ptr Timer;显示无敌风火轮
    Restart
    ;时钟中断结束时需要发送EOI信号给主从8259A
    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    iret
```

- **内核新增命令：可同时执行多个用户程序**

  根据实验要求可知，需要增加一条命令可同时执行多个用户程序，让内核加载这些用户程序，创建多个进程，实现分时运行。由于本实验实现了利用时钟中断进行多进程的调度执行，因此可以将之前的实验中的串行方式运行用户程序的做法替换成多进程的方式执行。

  为了增加这条新增命令的灵活性，因此不限定多进程的具体数量，数量可以从1~4变化，由用户输入自行指定，这样即使我们需要执行之前几个实验的如开关机程序等单个程序也可以利用这种多进程模型进行实现。由于涉及到输入命令字符串的比较、提取问题，因此这条新增命令的处理函数用C语言编写比较方便，C对于字符串的处理比较便捷简单。

  另外，由于在本实验实现的多进程调度执行的模型中，多进程的执行是由时钟中断处理程序中的调度函数以及Restart过程重启指定的，因此我们在内核程序中分析完指令后，只需将对应需要执行的用户程序加载入相应的扇区即可，而无需利用call指令跳转入程序中执行，因为跳转进入执行的操作都被囊括在时钟中断处理程序的调度、执行过程中了。

  - **加载用户程序函数LoadPro**

    因此不同于前面几个实验，我需要重新编写一个加载相应用户程序至内存相应位置的过程，在这个过程中，只需要将对应扇区的用户程序加载至内存相应位置即可。而由于在本次实验中，需要涉及多个用户程序的加载，为了合理安排内存空间，将各个程序以及对应内存空间安排为，内核程序放置在内存0x1000：0x100处；第一个输入的用户程序放置在段地址0x2000处，之后的三个用户程序依次递增0x1000，由于均为COM格式程序因此偏移量均为0x100。基于上述，在新编写的过程中，加载扇区至内存相应的段地址以及加载的扇区号，均作为过程的参数由调用过程传入，这样可以提高函数的通用性。

    根据前面几次实验的经验，我们只需要利用13h中断的02h读取扇区的功能即可实现。

    下面展示具体加载函数的代码：

```
public _LoadPro
_LoadPro proc
  push ax
    push bp;栈中sec/seg/ip/ax/bp

    mov bp,sp
mov ax,[bp+6]
    mov es,ax ;置es与cs相等，为段地址,入口时CS为0A00H
  mov bx,100h ;数据常量，代表偏移地址，段地址:偏移地址为内存访问地址
    mov ah,02h ;功能号02h读扇区
  mov al,1   ;读入扇区数
    mov dl,0   ;驱动器号，软盘为0，U盘和硬盘为80h
  mov dh,1   ;磁头号为1
```

```
    mov cl,[bp+8] ;起始扇区号，编号从1开始，传入的参数即为对应用户程序所在扇区，巨坑！！！！int\ip1\ax\ds\es\bp均在栈中！！！
cmp cl,8
    jz Par
    mov ch,0    ;柱面号为0
    int 13h     ;中断号
    pop bp
    pop ax
    ret
Par:
    mov al,4
    mov ch,0    ;柱面号为0
    int 13h     ;中断号
    pop bp
    pop ax
    ret

_LoadPro endp
```

特别说明地是，由于在实验五我们新增了一个测试封装的C库输入输出函数的用户程序test.com，该用户程序与其他四个用户程序只占一个扇区不同，该用户程序占用了四个扇区，因此在加载设置扇区数时需要单独讨论，具体如上。

- **新增命令的处理函数Load**

  由于新增了一条命令，形如“p+1234/324/...”。而“p"之后紧跟的数字代表加载的用户程序序号(也即扇区号)，数字数量代表同时执行的多进程数量，我们根据此命令来分析多进程模型下的若干个进程的加载以及内存区域的分配问题。

  首先，我们需要根据“p”之后的数字数量来给process_num(同时执行的多进程数量)变量赋值，其次，由于输入的数字也是对应用户程序的扇区号，因此我们需要依次将对应的用户程序加载进入相应扇区，等待时钟中断触发调度执行即可。

  下面展示具体代码：

```
/*分时并行执行时将需要并行执行的用户程序加载至相应内存*/
void Load(char *cmd)
{
    int i = 1;//除去p从下标1开始
    int num = 0;
    int count = 0;
    while (cmd[i] != '\0' && (cmd[i] >= 0+'0'&&cmd[i] <=9+'0'))
    {
        num = cmd[i] - '0';/*获得扇区号*/
        sec = num;
        LoadPro(segment,sec);
        segment += 0x1000;
        count++;
        i++;
    }
    process_num = count;/*当前准备并行执行的进程数*/
}
```

根据上述代码可以将对应的用户程序加载至相应内存区域。当然在，程序测试过程中发现一个巨坑，我们在时钟中断处理程序的设计中说到我们设定了一个总的调度次数用以控制多进程的同时执行的终止，有了这个总调度次数作为终止条件，那么在用户程序中我们就不需要利用所谓ret指令等返回内核程序了，我们只需要利用jmp $指令，让用户程序处于无线循环中等待总的调度次数达

到上限，返回内核即可。

在刚开始测试软盘时，我忽略了这一点，在多进程分时执行时，总是会出现程序刚开始可以顺利的分时执行，但一到快要结束执行时，就会出现黑屏的情况，后来通过反复查看用户程序代码以及结合Bochs单步调试才知道，原来是我的用户程序先于调度过程"擅自"返回内核程序，导致程序混乱出现黑屏的问题。后来，我将用户程序的返回指令用jmp $指令替换就成功解决了这个问题！

- **MyOS模块组织**

在本次实验中，MyOS的整个模块结构跟前面几次实验有较大区别，具体有以下几部分：

1. boott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：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,**新增多进程执行命令处理函数**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**、**设置时钟中断频率**、**修改时钟中断向量**、**新编时钟中断服务程序含Save**：中断处理的保护现场、**Restart**：中断处理的恢复现场函数以及旧的无敌风火轮显示处理程序。

   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、**多进程调度**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. 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。

另外，在本次实验中，合理规划了内存空间，具体分配如下：

内核kernel.com程序加载至内存0x1000:0x100处；

第一个COM格式用户程序加载至内存0x2000:0x100处；

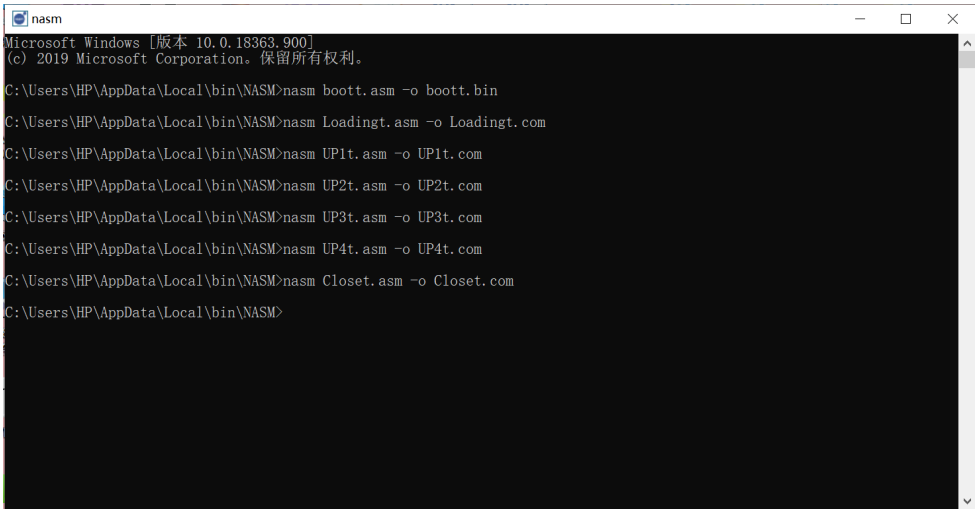
第二个COM格式用户程序加载至内存0x3000:0x100处；

之后以此类推！

- **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、testC库输入输出函数测试用户程序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。



2. TCC+TASM+TLINK链接编译

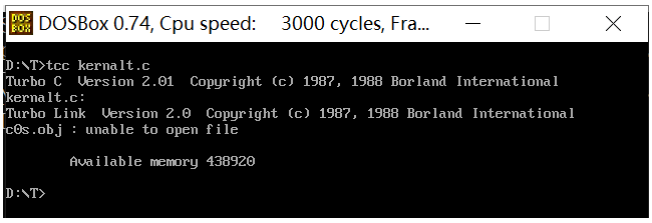
在DosBox中链接编译内核程序：rukout.asm、kernalt.c、bacist.asm、PCB.h

除此之外，还需链接编译C库输入输出函数测试test程序：Stuct.h、enter.asm、test.c、libs.asm

下面以kernel.com程序为例：

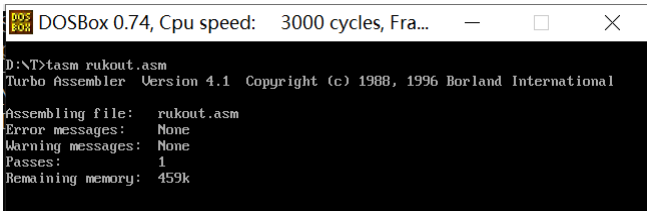
- **TCC编译**

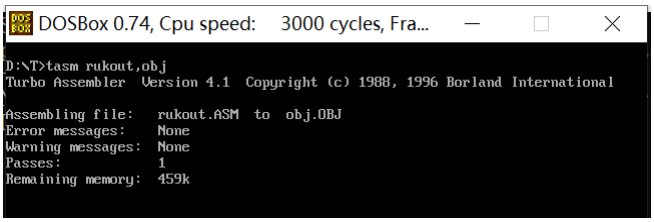
  用tcc+.c 形式生成同名.obj文件：



- **TASM编译**

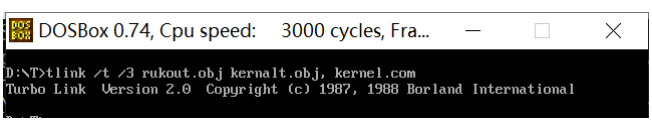
  用tasm+.asm形式先生成同名.obj文件，由于rukout.asm包含了basict.asm文件，因此还需要用tasm+rukout,obj形式再编译一次：





- **TLINK链接**

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

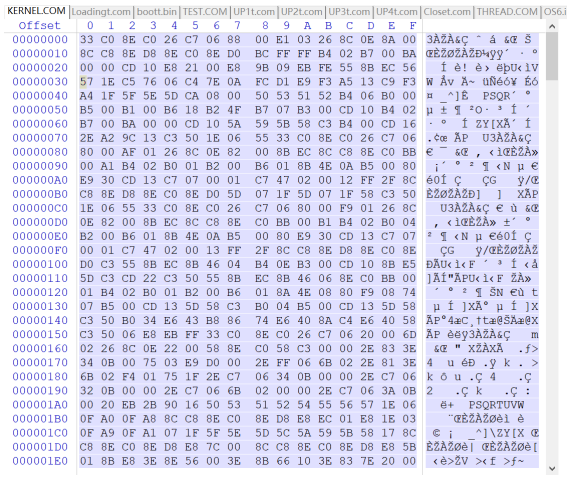


3. winHex工具编辑软盘

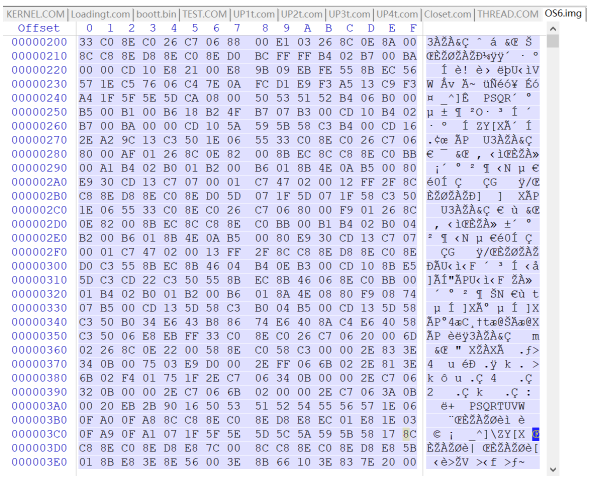
不同于实验一，在Linux环境使用dd命令进行1.44MB虚拟软盘映像的生成，在实验二、三、四、五中，我利用WinHex编辑软盘，两者相比较，我认为WinHex工具更为简便操作。因此，本次实验依然使用WinHex编辑软盘。

首先生成一个1.44MB的空白软盘OS6.img.利用winHex工具打开上述生成的各个二进制文件，先将“boott.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10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)、Loading程序、关机界面程序Close.com以及C库测试程序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：



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。

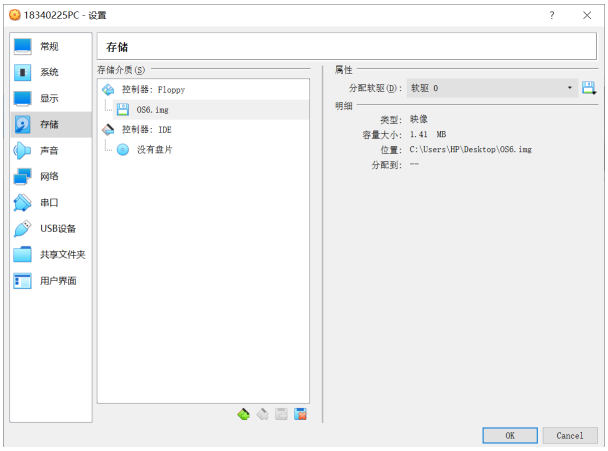


上图为1.44MB软盘，将上述”kernel“的内容复制到此软盘的第二个扇区开始的十个扇区中，操作为：右键——剪贴板数据——粘贴。

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。

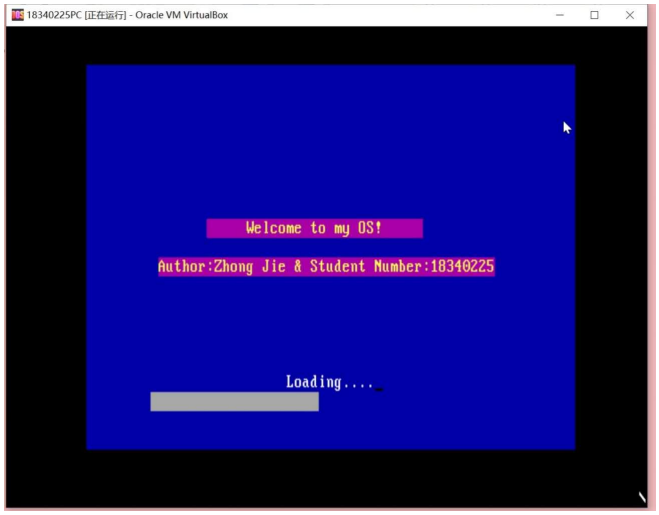


- **实验运行结果**

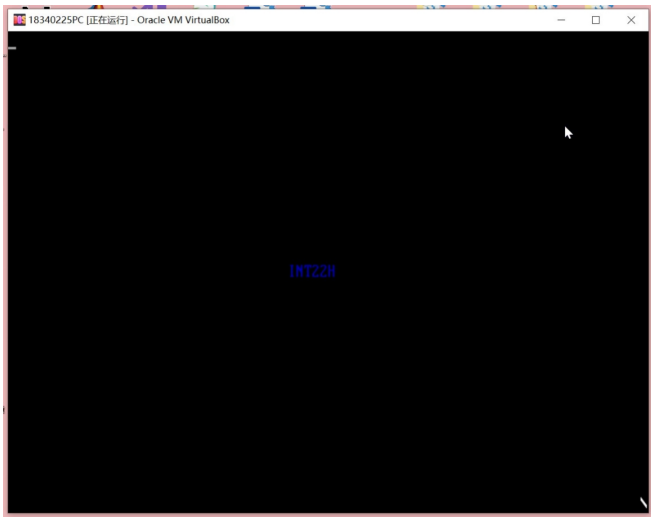
  成功启动虚拟机之后，虚拟机就会将程序加载到虚拟机的内存空间中运行，可以看到Loading程序的启动界面，也可以观察到内核程序的界面，接着输入help命令，依次按照菜单，输入选择运行不同的命令如：**p+想要同时运行的用户程序号**、dir、ls、batch、quit，同时执行不同的用户程序，查看用户程序信息、列出系统文件组织等功能，用户子程序的运行均以时钟中断触发的方式运行。同时还可以看到时钟中断响应程序的风火轮效果。

  程序运行结果如下：

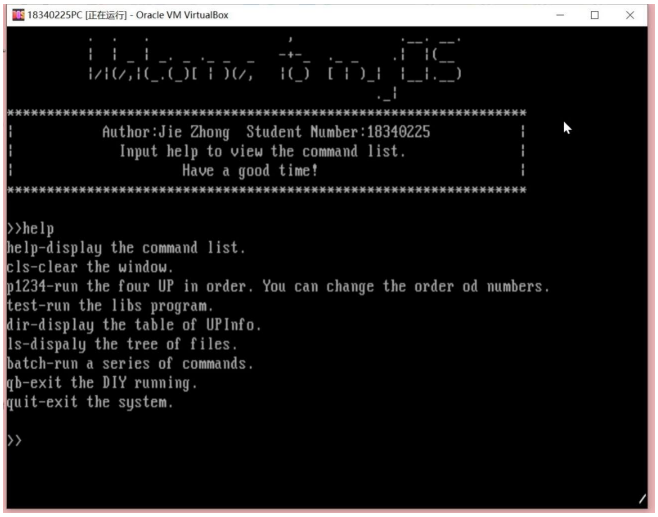
  Loading程序加载界面：



  进入内核界面，显性直接调用int 22h中断，显示“INT22H”字符串：

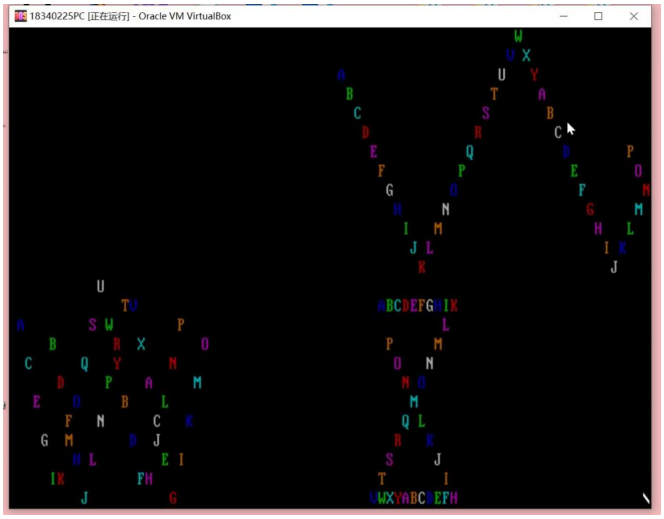


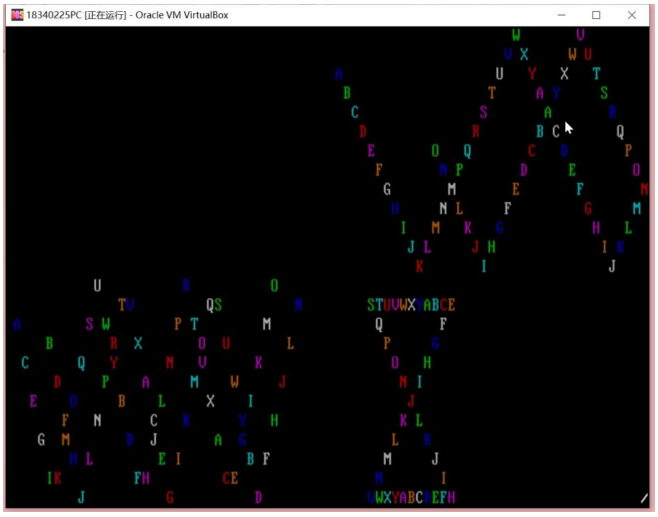
  内核界面，输入“help”，显示命令清单：



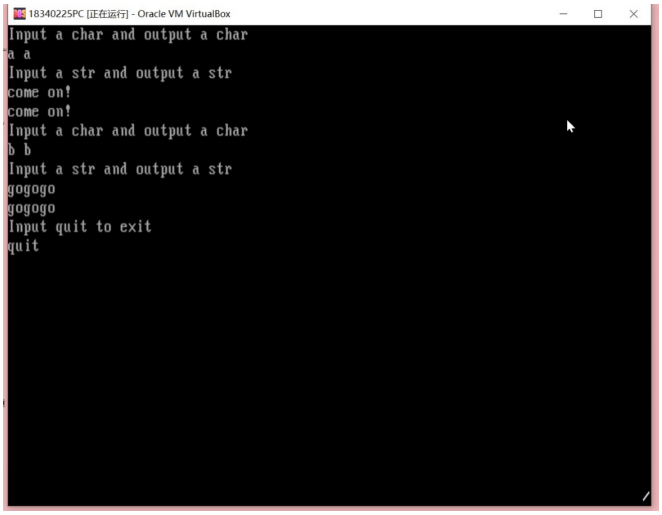
输入“cls”,系统执行清屏操作，接着输入“p1234”，同时执行四个用户程序：

到达总调度次数回到内核程序，接着输入“p234”，同时执行234三个用户程序：

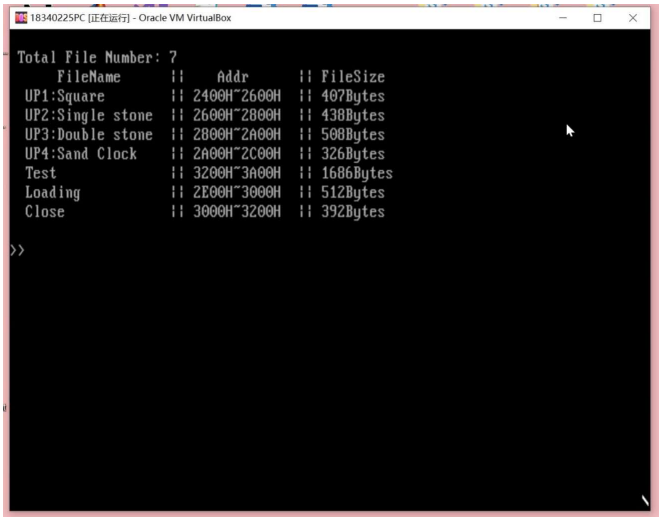




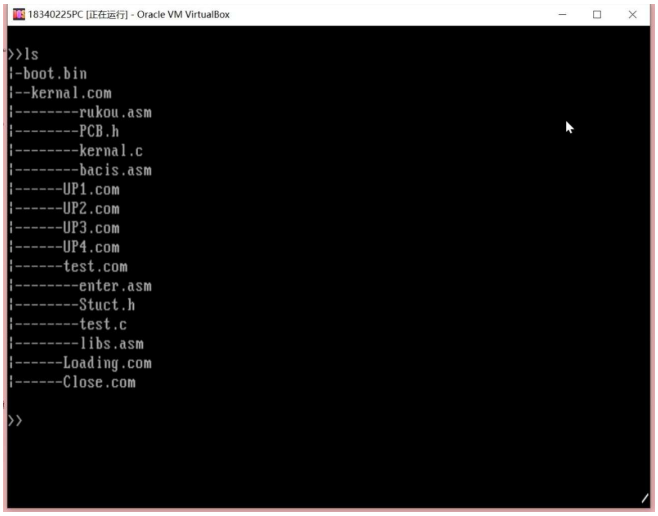
到达总调度次数回到内核程序，接着输入“test",显示新增的用以测试C库输入输出设计的用户程序：



到达总调度次数回到内核程序，接着输入”dir“，查看显示七个用户程序的表格信息：

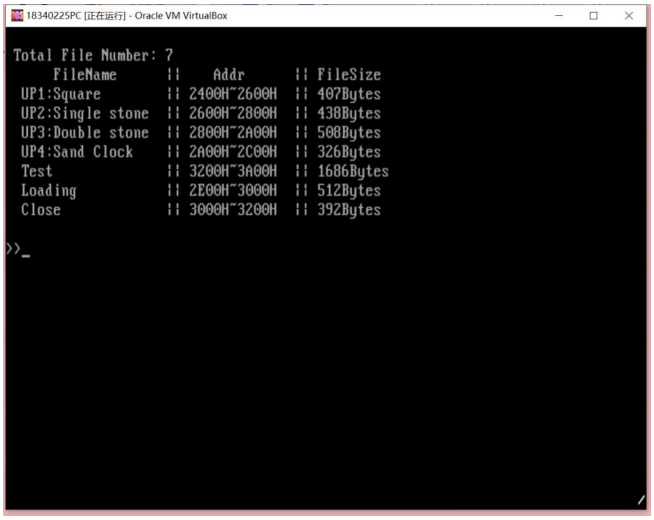


接着输入“ls”，查看此简易系统的文件组织：

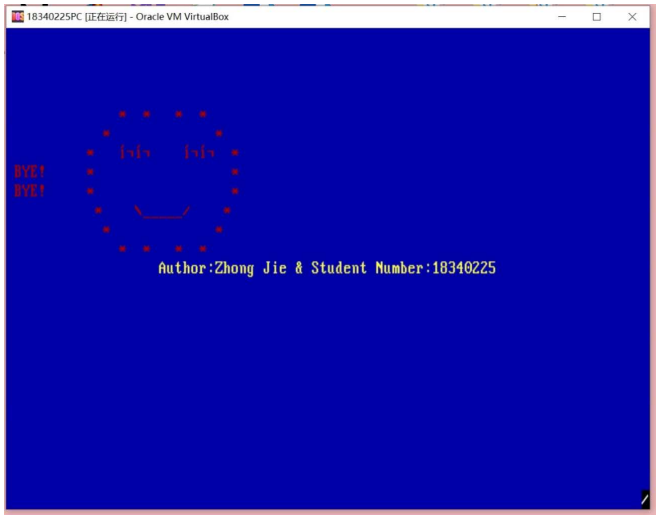


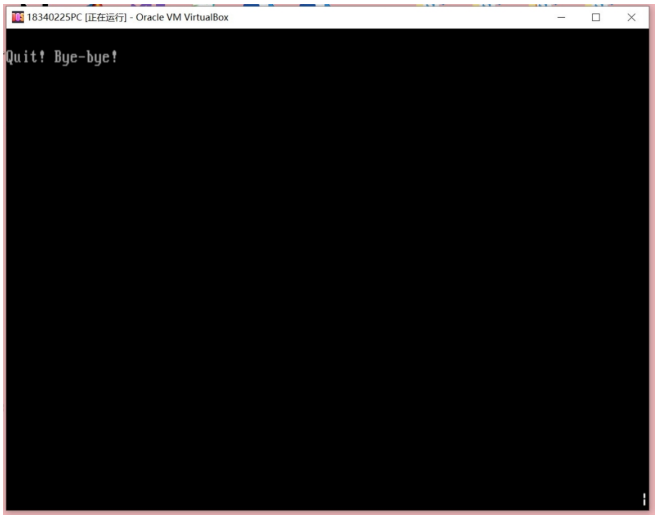
再输入“batch”，测试批处理命令，特别注意地是，由于在本实验中的程序执行方式变成并行方式，因此在批处理命令中，虽然命令是串行形式，但是其中每个命令的执行都是单进程下的并行执行！

测试执行一批“p123、p4、dir、qb"的指令：



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

本次实验的核心就是利用时钟中断结合实验五设计的中断处理的保护现场Save过程和恢复现场Restart过程实现多进程的并行执行。刚开始看到这个实验内容时，内心不免寒颤，但经过老师上课的详细讲述以及自己课后对于课件的反复理解。其实便可以将本次实验的技术关键概括为：对于实验五设计的Save和Restart过程的简单改进，设计进程控制块，设计多进程的调度函数，并且结合融入至时钟中断处理程序中！在实验五中，我们设计的Save和Restart过程是针对同一个进程而言的，而本次实验涉及到多进程的并行，因此Save和Restart过程在多进程模型下是针对不同进程的，Save过程是将上一个被中断进程的所有必要信息保存在对于的PCB结构中，而Restart过程则是将下一个被调度的进程的所有必要信息从对应的PCB结构中恢复至各个寄存器中。

而如何选择这所谓的下一个被调度进程呢？这与我们设计的多进程调度函数息息相关。根据实验要求，我需要设计一种命令，该命令指定多个进程分时执行，因此调度的顺序即是这条命令输入的进程顺序，我们按照这个顺序依次循环执行。当然在设计时有了这个大概思路之后，还需要将调度函数结合二状态模型（执行态和等待态）对于被中断进程和被调度进程赋予对应的状态。再结合老师所给的参考代码，对于多进程调度函数的编写相对来说比较简单，易于实现。

当然无论是Save和Restart过程，还是多进程调度函数的操作对象，均离不开进程控制块结构。在实验五中，我也简要设计了相关的结构，但由于实验五的要求比较简单，因此在设计PCB结构时，只考虑了所有上下文寄存器。而在本次实验中，设计多进程的调度问题，因此我们需要给予每个进程一个标志进程状态的变量，这样在调度过程中，我们才能清楚明了辨别各个进程的当前状态，防止出错。另外，实验五我们只涉及一个进程的保护和恢复，而本次实验中，在进入多进程并行状态后，由于时钟中断的触发，会涉及多个进程的保护与恢复执行，因此一个PCB控制块在一次多进程并行过程中是不能重复使用的，我们需要为多进程中的每个进程都分配一个进程控制块，这样在调度时，才能顺利恢复被调度进程的执行状态。除此之外，由于内核程序是myOS的整个核心，在执行完多进程并行后需要回到内核状态，因此我们需要为内核程序预留一个PCB控制块，不能被其他进程访问占用。

在理论课，我们需要了系统模式（或言内核模式）与用户模式，我们知道中断模式切换的契机，而本次实验也正是采用了这一点，利用时钟中断实现内核模式到用户模式的切换并且实现多个用户进程的切换。这与我们所学的理论知识相符合，但由于用户进程的切换执行均依赖时钟中断，因此如何结束多进程的并行是一个需要我们考虑的问题。在实验初期，我就是忽略了这个歌问题，因此在测试过程中，进行多进程的并行时，我的程序老师运行之后停滞在用户进程画面中，刚开始我很疑惑，后来利用Bochs单步调试，才知道原来是因为我没有设计多进程并行结束返回内核的操作，因此当用户程序执行结束后，不知何去何从。发现问题之后，我很快想到：时钟中断每触发一次，就进行一次多进程的调度，因此可以设计一个总的调度次数变量用以计算调度次数，当调度次数到达上限时，就返回内核状态。这点的考虑，终于让我的程序可以跑起来了！

当然，在本次实验中，实验五中遇到的Restart过程中的第一次运行与非第一次运行的问题，我仍然遇到了。这次我汲取了实验五的教训，没有与代码死磕，而是直接选择利用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。至于为什么会相差18呢，这应该是与实验五的原理一致的，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。有了实验五的前车之鉴，本次实验解决这个问题少走了很多弯路，实验就是一个不断积累经验的过程！

不知不觉，前六个基本实验已经接近尾声了，回顾一下前六个实验，从最开始的单个编写在引导扇区的用户程序，到现在能够实现多用户进程的并行的小型操作系统，整个过程十分不易，但也收获满满，在几个月前，我想都不敢想，自己可以写出一个属于自己的小型OS，这种感觉很奇妙！虽然Debug的过程很煎熬，但回过头看，一切都是值得的。接下来就要进入第七个实验开始的拓展实验阶段了，相比前六个实验，之后的拓展实验必将更加困难，但我会坚持下去，希望自己可以不放弃，努力Debug，为自己的OS再润色！

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=distribute.pc_relevant.none-task-blog-baidujs-1