实验六：时间片轮转的二态进程模型

1 实验题目

实现简单的多进程。用时间片轮转的方法交替执行用户进程，进程为二状态进程模型。
具体来说，共有五项：

1. 修改实验五的内核代码，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，再定义一个进程表数组，最大进程数为 10 个；
2. 扩展命令处理，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。

2 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。

3 实验要求

1. 内核一次性加载多个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占 1/4 屏幕区域，信息输出有动感，以便观察程序是否在执行；
2. 在原型中保证原有的系统调用服务可用。

4 实验方案

工具与环境：Windows（x86_64），gcc，NASM，ld 2.25.1，bochs 虚拟机（1MB 内存、1.44MB 软驱）

本次实现的“时间片轮转实现多进程”的原理是，多个进程同时加载到内存的不同位置，执行某一个进程一段时间后，利用 `int 08h` 系统时钟中断来切换进程，使得用户进程轮流执行，看起来好像是同时进行一样。

之前实现的四个射字母程序正好位于屏幕的四个区域，因此用它们来测试，若屏幕多个区域同时射字母，则表示成功。

多进程要用的进程控制块、CPU 上下文保护、系统时钟中断处理等数据结构及方法，都在前面的实验中部分实现了，因此本次实验的重点是完善之前的数据结构及方法，使系统支持多进程。难点是进程信息的保护和恢复不能出错、系统再入的处理、多进程管理的规范。

结合布置的实验内容，我将按照如下步骤进行：（顺序为从 C 到汇编、从结构设计到功能使用）

1、添加相关数据结构及变量 包括进程控制块 PCB、进程表、内存管理等；

2、修改命令解释器部分 增加多进程命令设计；

3、编写进程调度过程 用于寻找下一个执行的进程；

4、修改 `save`、`restart`、`int 21h` 所用的现场保护及恢复 信息的保存从上次的临时内存变量迁移到相应的进程控制块中；

5、修改系统时钟中断 调用风火轮及进程调度函数，使之成为完整的进程切换；

6、修改“加载用户程序”“结束用户程序”部分的代码 使之适配多进程；

7、测试 用射字母程序来测试多进程。

实验过程部分将以流畅的方式叙述实验细节，更多试错过程写在实验总结部分。

5 实验过程

5.1 添加相关数据结构及变量

描述一个进程的基本数据结构是进程控制块，内容包括进程信息（进程 id、进程名、进程状态）、CPU 上下文：

```
typedef struct cpuState{
        int eax,ebx,ecx,edx,ebp,edi,esi;
        short ip,cs,flag,ds,es,ss,sp;
} cpuState;

typedef struct PCB{
        cpuState cpuS;
        char pid,pname[10],state;       // state: 0--empty; 1--running; 2--ready
} PCB;
```

由于是二状态进程模型，故 state 只安排两个有意义的值——1 表示进程正在运行，2 表示进程就绪。此外，0 表示进程表该元素为空。

注意安排各变量顺序，以避免 struct 内存对齐引起不必要的浪费。

接下来是多进程管理需要用到的变量，包括一个进程表 proList、就绪队列 readyList、就绪队列的头尾指针、当前进程标号 curPid、pid 计数器 pidTotal：

```
#define MAXPRO 10
#define MAX_RL_LEN 16
#define MOD_RL_LEN 15

PCB proList[MAXPRO+1];
char pidTotal;
int readyList[MAX_RL_LEN],rlHead=1,rlTail,curPid=MAXPRO;
```

进程表是一个用于储存各进程 PCB 的数组，大小为 11，其中 0~9 存储用户进程的 PCB，10 存储系统再入时的内核 CPU 上下文（后文解释）。

此处实现一个简单的内存管理，即进程表的 10 个用户进程项分别对应内存从 0x2000:0x0000 开始的 10 个 32KB 段。当需要加载执行用户程序时，在进程表中找到空项，就将用户程序加载到那一项对应的内存中去。这是一个较粗糙的“直接对应法”，但优点是方便管理。

多进程模型需要一个管理就绪进程的队列，以实现轮流执行进程。此处使用进程在进程表中的下标作为队列元素，这是因为 PCB 是比较大的内存结构，把一个 PCB 从对头添加到队尾会产生较大的内存开销，而以下标作为队列元素，入队出队只需移动一个 int。

就绪队列为循环队列，长度为 MAX_RL_LEN=16，取该值有两个原因：其一，它大于最大进程数，因此可以使用 rlHead=(rlTail+1) mod MAX_RL_LEN 来判断队列为空；其二，选择一个 2 的幂作为模数，能够使得 $x \bmod 16 = x \text{ and } 15$，即用位运算代替模运算，大大提升运算效率。

当前进程号 curPid 表示当前运行的进程（值为其在进程表中的下标），这主要是给汇编保存上下文相关过程使用的，特别地，如果 curPid=MAXPRO 则表示是内核态。

计数器 pidTotal 是给新产生的进程赋予进程号的，按照现代操作系统的做法，每个进程会有独一无二的 pid，即便是生存期不相交的进程，进程号也不相同。因此采用一个单调递增的计数器来实现。

5.2 修改命令解释器部分

此处简单实现一种命令格式为：不同的用户程序名用“|”隔开，表示同时运行，例如“1|2|3”。

判断为这种命令之后，执行的伪代码如下：

```
for(int i=0; i<buflen; i+=2) if (buf[i]>='1' && buf[i]<='5')
{
        int id=buf[i]-'0', st=id+11, seg_addr;
        for(int j=0; j<MAXPRO; j++) if (proList[j].state==0)
        {
                // update proList
                // here modify proList[j], set seg_addr=0x2000+0x800*j

                // update readyList
                rlTail=(rlTail+1)&MOD_RL_LEN;
                readyList[rlTail]=j;
```

```
12
13                  // load_client
14                  for(short num=sectorNum[id], bx=0x100, ch=st/36, dh=(st/18)&1, cl=(st%18)+1;
15                          num;
16                          num--, bx+=0x200, ch+=(dh==1 && cl==18), dh^=(cl==18), cl=(cl%18)+1)
17                                  load_client(seg_addr,bx,dh<<8,(ch<<8)+cl);
18                  client_preparation(seg_addr);
19
20                  break;
21          }
22  }
23  curPid=readyList[rlHead];
24  restart();
```

该段逻辑为：扫描 buf 获得需要加载的进程，寻找进程表的空项，找到后，更新进程表项、就绪队列，加载用户程序到内存并做初始化准备，等所有用户程序完成准备工作后，使用 restart 过程来开始用户进程。

更新进程表项的部分，要对 CPU 信息赋初值（置段寄存器值为该表项所对应的内存段基址，即 seg_addr，以及 ip,sp,flag 改为恰当的值，其余置 0），这样后面就可以直接使用 restart 来开始这个进程了。并且修改 pid,pname，置 state 为 2。

加载用户程序到内存的部分，由于现在文件已经要储存在软盘较深入的地方，加载过程中磁头号、柱面号都可能要变化。此处实现一个自动加载过程，给定初始扇区号、扇区数，自动地一个一个扇区地加载到内存。

当所有准备工作完成后，初始化 curPid，并调用 restart 开始第一个进程（curPid 指向的进程）。

5.3 编写进程调度过程

进程调度过程的任务是，当中断（包括 int 08h,int 20h）发生后，系统用来切换进程，获知下一个执行的进程。该过程的主要逻辑很简单，就是更新队列然后取就绪队列的队头元素。

代码如下：

```
1  void ProSchedule()                          // process scheduling
2  {
3          if (curPid==MAXPRO) return;
4          if (proList[curPid].state==0) rlHead=(rlHead+1)&MOD_RL_LEN;
5                  else if (proList[curPid].state==2)
6                  {
7                          rlTail=(rlTail+1)&MOD_RL_LEN;
8                          readyList[rlTail]=readyList[rlHead];
9                          rlHead=(rlHead+1)&MOD_RL_LEN;
10                 }
11         curPid= (rlHead==((rlTail+1)&MOD_RL_LEN)) ?MAXPRO :readyList[rlHead] ;
12 }
```

首先，该过程是中断发生后被调用的，因此先要判断当前进程 curPid 发生了什么——curPid=MAXPRO 表示原本在内核态，此时不可以执行进程调度（下文述原因），退出；state=0 表示是用户进程结束后调用 int 20h 进来的，此时队头出队表示删除进程；state=2 表示是其他中断导致用户程序暂停，此时为了轮流执行别的进程，当前进程要从队头移到队尾。

然后更新当前进程 curPid，若队列为空则赋值为 MAXPRO 表示返回内核，否则赋值为队头元素。

内核态时不执行进程调度，原因如下：

1. 若由普通的内核过程（命令解释器期间等）进入 int 08h，此时没有其他进程可调度；
2. 考虑到二状态进程模型的缺点，若用户程序调用 int 21h 进入内核，其实并不处于就绪态，而是阻塞态，等待系统资源。而此时如果切换进程，则有可能将这个未得到资源的用户程序重新运行起来，产生错误；
3. 在上一点情形下，由于系统各服务未独立成进程或线程，这也将使得正在进行的系统调用服务丢失。

5.4 修改现场保护及恢复

接下来轮到汇编的部分。之前已经实现了 save 和 restart，不过当时并没有进程表，因此在内存开了一个暂用的变量 tmpCpuState 来储存 CPU 上下文。现在只需把这个储存地址改为进程表项即可。

所要保存的位置为 proList[curPid].cpuS，首先需要用一个寄存器 bx 来计算得到这个地址，代码为

```
save:                           ; protection
        push ebx
        push eax
        push ds
        mov ax, cs              ; use OS's ds
        mov ds, ax
        mov ebx, [_curPid]      ; bx = &proList[curPid]
        imul bx, 56             ; sizeof(cpuState)=56
        add bx, _proList
        ...
```

这样就得到 [ds:bx] 表示 proList[curPid].cpuS 的地址了，就可以保存上下文了。

同时，save 的最后还要修改 proList[curPid].state=2。

restart 和 int 21h 同理，进行正常工作前算出 ds:bx 表示 proList[curPid].cpuS 的地址。restart 的最后，跳转到用户程序前，要修改 proList[curPid].state=1。而在 int 21h 的保护现场工作中，还要更换 curPid 为 MAXPRO，表示进入了内核态：

```
mov ebx, [_curPid]
mov dword [clientPid], ebx      ; save curPid
mov dword [_curPid], MAXPRO     ; claim that we are in OS
```

同理，离开前执行逆过程恢复 curPid。这样，系统服务期间若发生系统时钟中断，则 CPU 上下文就保存到了 proList[MAXPRO].cpuS，可以正常执行风火轮，且不发生进程调度。

此处修正以前程序的一个 bug，内核的 bp 寄存器也要保存，像 os_sp 一样，也开一个内存变量 os_bp 用于保存该寄存器。

5.5 修改系统时钟中断

重新编写一个过程作为系统时钟中断处理。该过程目前只有两个功能——风火轮和进程调度，因此代码逻辑为依次执行 save,HotWheel,ProSchedule,restart，代码如下：

```
Timer:
        call save
        HotWheel:
                ; here HotWheel
        Call_ProSchedule:
                cmp dword [_curPid], MAXPRO     ; if no client process, exit
                jz Timer_End
                push word 0
                call _ProSchedule
Timer_End:
        mov al, 0x20                            ; tell 8259 that we finished
        out 0x20, al
        jmp _restart
```

注意此处有一个判断，如果 curPid=MAXPRO 则不进入进程调度。这看似与 5.3 节进程调度过程的第一句话重复了，那么这个判断有必要吗？是有的，因为 call 这个过程要用栈，而如果发生了系统再入，必须小心谨慎不能碰栈，因此连进都不能进去。事实上，这个判断才是真正起作用的，5.3 节的判断只是双保险，增加鲁棒性。

5.6 修改“加载及结束用户程序”部分的代码

加载用户程序分为“将用户代码加载到内存”和“进入用户前的准备”两部分。对于前者，改成适配 5.2 节的加载方式即可（即功能寄存器的值从参数中获得）；对于后者，原本这里会修改段寄存器、栈、跳转到用户程序，但现在这些功能已经在 5.2 节实现了，因此这里只保留“在用户程序 PSP 字段埋 int 20”“用户栈开头埋下 0x0”这两个功能：

```
global _client_preparation
_client_preparation:
        mov ax, [esp+4]
        mov es, ax
        mov word [es:0], 0x20cd         ; int 20h
        mov dword [es:0x7ffb], 0        ; push dword 0
        o32 ret
```

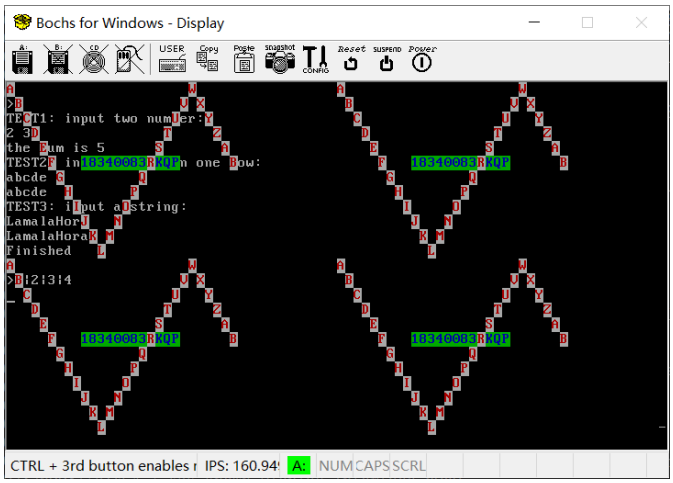
结束用户程序的 int 20h 过程，共三个任务：恢复内核的上下文、置 proList[curPid].state=0（表示结束该进程）、进程调度：

```
end_client:
        ; here recover ds,ss,sp,bp
        mov ebx, [_curPid]                              ; bx = &proList[curPid]
        imul bx, 56
        add bx, _proList
        mov byte [ds:bx+55], 0                  ; set PCB state to 0--empty
        push word 0
        call _ProSchedule                               ; check if client processes exist
        cmp dword [_curPid], MAXPRO
        jnz _restart
        o32 ret
```

当前进程会在进程调度过程中被移出就绪队列。若进程调度的结果是 curPid=MAXPRO，则表示所有进程结束，此时 ret 返回命令解释器（即 ker 过程）；否则，restart 继续执行下一个进程。

5.7 测试

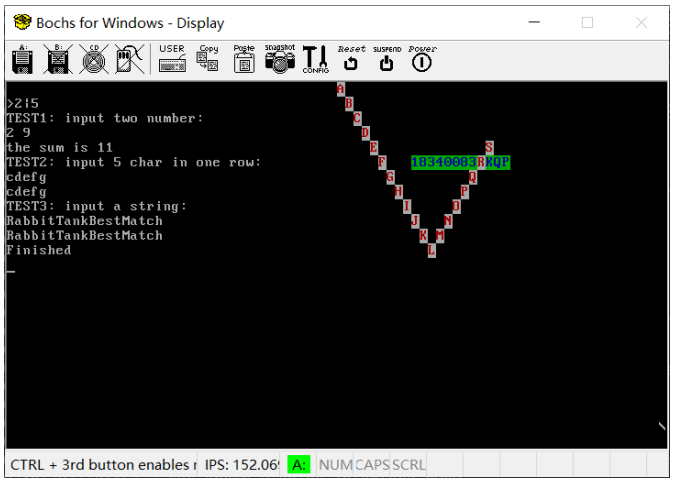
至此，所有过程编写完毕。先后采用库测试程序、四个射字母程序进行测试，效果如下：（助教可运行 myOS.bxrc 查看动态效果）



可以看到：

1. 库测试程序输入输出都正常，说明 `int 21h` 相关过程正确；
2. 输入输出期间（包括内核命令输入、库测试程序输入），风火轮照常转动，说明 `int 08h` 功能正常；
3. 四个射字母同时运行，速度明显减慢，说明多进程成功，分时复用正确。

当然，也可以让库测试程序和射字母程序同时运行，这样的效果是，射字母程序完全在库测试程序之后运行。这是因为库测试程序多数时间在等待输入，而我们禁止了系统调用服务期间切换进程。



6 实验总结

这次实验不算是有新的技术，但是是对原有技术的一次大整合，使其功能上升了一个等级。所以前序实验的铺垫很重要。

6.1 体会

有两点体会较深刻。

一是这次试验相比上次，代码改动还是较大的，很多地方都要修改、重写、新增内容，这时拥有一个修改的顺序、修改的逻辑很重要。我的修改顺序是“由 C 到汇编、由结构设计到功能使用”，这样整体逻辑很顺畅，且避免东写一点西写一点，会丢失很多细节，增加 debug 难度。

二是现在的代码长度都很大了，内核有 9 个扇区，调试起来相比实验二的时候已经是困难了许多倍。所以需要讲究调试的方法技巧，比如提升汇编阅读能力以快速地找到 C 代码对应的汇编代码位置、记住一些内存位置快速地断点过去、较短的代码优先用肉眼读。

6.2 思考的问题

最主要思考的还是系统再入的问题。上一次实验思路是全面禁止系统再入，但实际上没禁完，命令解释器期间仍然可以转风火轮。后来仔细思考，是因为这时候的风火轮既没有覆盖 CPU 上下文保存，也没有使用栈，因此虽然再入了，但是没有影响。

于是这次萌生了一个想法：是否可以允许小部分的系统再入？例如这类不需要栈的功能？

事实上如果按这个思路来做，那么这个允许再入是很鸡肋的——除了风火轮哪有什么不需要栈的功能！但是完全 ban 掉系统再入，又无形中增加了枷锁，输入期间不转风火轮令人不爽。

最终通过与同学的交流，得到了一种很好的思路——允许再入，在进程表中留一项给内核存放再入前的 CPU 上下文，禁止内核态使用进程调度。

如此一来，在 `Timer` 过程中给内核态留了一条完全不用栈的通道，可以在任何时刻都把风火轮转起来了。只不过，“不能用栈”这个鸡肋的设定，依然需要通过后面的多线程、信号量、系统服务独立成进程来破除。

6.3 试错细节

一些常见的错误包括：

1、栈的对齐、栈的顺序，C 代码栈是 4 字节对齐的，即便传入的是 `char,short` 类型；

2、`bp` 和 `sp` 一样是需要保存的，它们一起实现了栈功能，但前者往往被忽略。

参考文献

[1] 田宇,一个64位操作系统的设计与实现,人民邮电出版社