

实验六：实现时间片轮转的二态进程模型

计算机科学与技术 18340146 宋渝杰

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

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。

实验过程：

操作环境：

- 操作系统：win10 + Linux
- 虚拟机：VirtualBox
- C 编译器：gcc
- x86 编译器：nasm
- 链接器：ld
- 本次实验新增调试工具：Bochs

步骤一：实现 PCB

在实验五实现 `save()` 和 `restart()` 的过程中，我已经在内核的 `c` 模块建立了结构体 `PCB`，已经拥有了 CPU 寄存器保存区，用于实现 `save()` 和 `restart()` 的功能。而这次实验我根据实验要求，在原有的结构体 `PCB` 内添加了进程号、程序名、进程内存地址信息、进程状态，共 4 个数据项。同时声明结构体数组，大小为 10，声明两个指针 `qi` 和 `qe`，分别用于进程的切换和进程的写寄存器操作：

```
struct PCB {
    // pid: 进程号, statu: 进程状态, exename: 程序名, address: 进程内存地址信息
    // q[10]: 结构体数组, qi: 进程切换, qe: 进程写寄存器
    int ip, cs, flags, es, ds, ss, ax, bx, cx, dx, di, bp, sp, si, pid, statu;
    char exename[10];
    char address[10];
} q[10], *qi = q, *qe = q+1;
```

步骤二：新增命令，实现进程并行

由于我之前的命令功能均在 `c` 模块实现，因此只需在我原有的 `c` 模块实现命令功能的代码上稍微加几行，就可以实现命令的新增。我这次的并行命令命名为 "runs"。

新增了命令之后，之后就是并行功能的实现。由于**并行功能实现的基本原理，是寄存器值的切换**，因此我总体的设计思路如下：

- `c` 模块：识别指令 `runs`，然后在 `PCB` 中的寄存器位置写入用户程序（反弹程序）的初始寄存器值
- 时钟中断：在 `c` 模块写好用户程序初始寄存器值之后，定时调用进程调度函数，切换寄存器值
- 进程调度函数：也写在 `c` 模块内，对步骤一的 `qi` 指针进行地址切换，配合 `save()` 和 `restart()` 实现寄存器值的切换
- `save()` 和 `restart()`：先通过 `save()` 将当前运行程序的寄存器值保存在旧的 `qi` 指针指向的 `PCB` 寄存器保存区的位置，在进程调度函数切换了 `qi` 指针，指向新的 `PCB` 寄存器保存区之后，通过 `restart()` 写入新的寄存器值

显然，这四步实现思路分别对应了本次实验的实验内容二到实验内容五，因此，分别在对应的步骤中阐述实现的思路和过程。那在步骤二这里，主要介绍指令的识别和用户程序初始寄存器值的写入：

指令的识别：由于我之前编写了 `c` 语言的字符串匹配函数 `cstr()`，因此只需建立一个静态字符串 `const char run[] = "runs";`，并在指令匹配的时候调用该函数，函数返回 0（完全匹配）即完成指令的识别。

```
const char run[] = "runs";

int cstr(const char *s1, const char *s2) { // 字符串匹配函数
    while (*s1 && (*s1 == *s2))
        ++s1, ++s2;
    return (int)*s1 - (int)*s2;
}

else if (cstr(str, run) == 0) { // 匹配成功
    ... // 写入用户程序（反弹程序）的初始寄存器值
}
```

用户程序初始寄存器值的写入：这里我的思路是通过调试工具 `Bochs`，在内核刚刚成功调用用户程序，用户程序即将要执行第一步时，获取所有的寄存器 `ip, cs, flags, es, ds, ss, ax, bx, cx, dx, di, bp, sp, si` 的值，即为用户程序初始寄存器值（`flag` 不采用当时的值，而特别修改为 512：中断允许 `IF=1`），之后写给用户程序对应的 `PCB` 寄存器保存区的位置即可

下图为用户程序一的初始寄存器值：

```
(0) [0x00000000a300] 0000:a300 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:3> r
rax: 00000000_00000005
rbx: 00000000_00001f40
rcx: 00000000_00000000
rdx: 00000000_00000000
rsp: 00000000_0000fba0
rbp: 00000000_0000fba0
rsi: 00000000_000e008a
rdi: 00000000_0000ffac
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
r12: 00000000_00000000
r13: 00000000_00000000
r14: 00000000_00000000
r15: 00000000_00000000
rip: 00000000_0000a300
eflags 0x00000002: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af pf cf
<bochs:4> sreg
es:0xb800, dh=0x0000930b, dl=0x8000ffff, valid=7
    Data segment, base=0x000b8000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=3
    Data segment, base=0x00000000, 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=0x000000000000f9af7, limit=0x30
idtr:base=0x0000000000000000, limit=0x3ff
```

将图中有效信息提取，形成代码如下：

```
int muti=0; // 标识变量：用于表示用户程序初始寄存器已写好，可以开始并行

else if (cstr(str, run) == 0) {
    for (int i=0; i<4; i++) { // 写 4 个用户程序的初始寄存器值
        (qe+i)->flags = 512;
        (qe+i)->ip = 0xa300+i*0x200; // 用户程序不同，ip 不同
        (qe+i)->sp = 0xfba0;
        (qe+i)->bp = 0xfba0;
        (qe+i)->si = 0x000e008a;
        (qe+i)->di = 0xffac;
        (qe+i)->ss = 0x0000;
        (qe+i)->es = 0xb800;
        (qe+i)->ds = 0x0000;
        (qe+i)->cs = 0x0000;
        (qe+i)->ax = 0x0005;
        (qe+i)->bx = 0x1f40;
        (qe+i)->cx = 0x0000;
        (qe+i)->dx = 0x0000;
        (qe+i)->statu = 1;
        ...
        LoadnEx(i+1); // 加载用户程序到指定内存位置
    }
}
```

```

    }
    muti = 1; // 允许开始并行
    while (muti) {} // 内核死循环，暂停内核工作
}

```

至此，已经基本完成 "识别指令 runs，然后在 PCB 中的寄存器位置写入用户程序（反弹程序）的初始寄存器值" 的功能

步骤三：修改时钟中断

这部分较为简单，只需要加入进程调度函数的调用即可，具体的进程切换在进程调度函数中实现

另外有一个需要注意的点，只有上一步骤的标识变量为 1 时，才允许进程的调度

```

int8h:
    cli                    ; 屏蔽外部中断
    call save
    push 0
    call draw              ; c 模块“数字钟”显示

; 下面是新增部分
    cmp word[muti],0       ; 判断是否允许进程调度
    jz re                  ; 不允许进程调度
    push 0                 ; 压栈，服务 c 函数返回
    call schedule           ; 调用进程调度函数
re:
; 新增至此

    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al            ; 中断结束
    pop ax
    sti                    ; 解除屏蔽
    jmp restart

```

步骤四：进程调度函数

这部分也较为简单，由于 c 模块已经定义了结构体 PCB 和指针 qi，因此进程调度函数只需对指针进行切换，切换到下一个运行的用户程序的结构体 PCB 寄存器保存区位置即可。

```

void schedule() {          // 进程调度函数
    if (++qi == q+5) qi = q; // ++qi，到末尾了就返回头部
                             // 由于内核和用户程序数量为 5，因此末尾定为 5
    while (qi->statu == 0)  // 如果该用户程序为非就绪态，则再切换到下一个
        if (++qi == q+5) qi = q;
}

```

步骤五：修改 save() 和 restart()

这部分本质上无需修改，`save()` 会将进程寄存器数据保存在旧的指针位置，而进程调度函数已经将指针定位到新的位置，因此 `restart()` 函数会读取新的位置（即新的程序）的寄存器值，赋值到寄存器后中断返回，即跳转到下一个用户程序执行

`save()` 和 `restart()` 实验五代码：

```
save:
    push ds
    push cs
    pop ds                ; ds 指向内核
    pop word[save_ds]     ; 保存了原始的 ds，即用户程序
    pop word[save_cs]     ; 保存了 save 返回的地址
    mov word[save_si],si
    mov si,word[qi]       ; c 中的 save 结构体
    pop word[si]          ; ip
    pop word[si+4]        ; cs
    pop word[si+8]        ; flags
    mov word[si+12],es    ; es
    push word[save_ds]
    pop word[si+16]       ; ds
    mov word[si+20],ss    ; ss
    mov word[si+24],ax    ; ax
    mov word[si+28],bx    ; bx
    mov word[si+32],cx    ; cx
    mov word[si+36],dx    ; dx
    mov word[si+40],di    ; di
    mov word[si+44],bp    ; bp
    mov word[si+48],sp    ; sp
    push word[save_si]
    pop word[si+52]       ; si
    jmp word[save_cs]

restart:
    mov si,word[qi]
    mov es,word[si+12]    ; es
    mov ss,word[si+20]    ; ss
    mov ax,word[si+24]    ; ax
    mov bx,word[si+28]    ; bx
    mov cx,word[si+32]    ; cx
    mov dx,word[si+36]    ; dx
    mov di,word[si+40]    ; di
    mov bp,word[si+44]    ; bp
    mov sp,word[si+48]    ; sp
    push word[si+8]       ; flags
    push word[si+4]       ; cs
    push word[si]         ; ip
    push word[si+52]
    push word[si+16]
    pop ds                ; ds
    pop si                ; si
    iret
```

额外步骤：os 优化

由于加入了新的功能，此处需要对内核以及用户程序要有相应的必要调整

用户程序：由于用户程序的多进程同时运行需要将不同的用户写入不同的内存地址，并进行内存地址（实际上即 ip）的切换，因此用户程序的地址定位代码需要修改为不同的代码。例如，用户程序一的定位代码修改为 0xA300，用户程序二的代码修改为 0xA500，以此类推

内核汇编模块：一个要修改的是加载内存函数 LoadnEx，需要把不同的用户程序加载到不同的内存位置（而之前固定加载在 0xA100），因此需要做稍微的修改，根据传入的参数（用户程序"几"）加载到对应的地址：

```
LoadnEx:
    push ebp                ; ebp入栈
    mov ebp, esp            ; 因为esp是堆栈指针，无法暂借使用，所以得用ebp来存取堆栈
    mov ecx, [ebp+8]
    mov ax, cs               ; 段地址：存放数据的内存基地址
    mov es, ax               ; 设置段地址（不能直接mov es,段地址）
    mov bx, OffsetOfUserPrg1 ; 偏移地址；存放数据的内存偏移地址
    mov ch, 0                ; 柱面号：起始编号为0
    mov ax, 200h
    mul cx
    add bx, ax               ; 根据传入参数 cx 修改为新地址
    mov word[jmpAdd], bx
    mov ah, 2                ; 功能号
    mov al, 5                ; 扇区数
    mov dl, 0                ; 功能号
    mov dh, 1                ; 磁头号：起始编号为0
    int 13h                  ; BIOS的13h功能
    call cls                 ; 调用清屏函数
    cmp word[judge], 0
    jz bak
    jmp word[jmpAdd]
bak:
    mov esp, ebp
    pop ebp
    ret
```

另一个是 int 20h 需要做些修改，在用户程序中断返回时，修改并行标识变量为 0，终止多进程并行

```
int20h:
    call save
    call cls
    mov word[muti], 0        ; 新增代码：结束并行
    jmp main
    jmp restart              ; 结构对称，实际上该代码不执行
```

内核 c 模块：第一点是在并行结束，返回内核程序时，需要将 qi 指针重新置为结构体数组头部；第二点是更新 help 指令的提示字符串，加入对 runs 的提示字符串

```

    qi = q;
    ...
static char helpstr[] =
    "You can input these instructions.\n"
    "\n"
    "help      -- Print instructions\n"
    "exe       -- Open an exe\n"
    "exes      -- Open many exes\n"
    "runs      -- run all basic exes at the same time\n" // 加入这行
    "ls       -- Show four exes' information\n"
    "cls       -- Clear the screen\n"
    "exit      -- Exit OS\n";

```

除了一些必要的调整，我也对实验五的代码进行了一些优化，主要是时钟的显示，原本是在内核时才显示时钟，现在修改为在用户程序（串行以及并行）也可以显示

```

void draw() {
    if (1) {    // 从 judge == 1 改为 1
        ...    // 显示时钟
    }
}

```

Linux 下运行以下编译命令行，生成 6-1.img 文件，参数和之前的实验一样，这里不再赘述

```

// 内核
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 6-1c.c -o 6-1c.o
nasm -felf 6-1asm.asm -o 6-1asm.o
ld -m elf_i386 -N --oformat binary -Ttext 0x7e00 6-1asm.o 6-1c.o -o 6-1.bin
nasm 6-1os.asm -o 6-1os.bin

// 用户程序
nasm 1.asm -o 1.com
nasm 2.asm -o 2.com
nasm 3.asm -o 3.com
nasm 4.asm -o 4.com
nasm 5.asm -o 5.com
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c io.c -o io.o
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 6.c -o 6c.o
nasm -felf 6.asm -o 6asm.o
ld -m elf_i386 -N --oformat binary -Ttext 0xa100 6asm.o 6c.o io.o -o 6.com

// 生成 6-1.img 文件
rm -f 6-1.img
/sbin/mkfs.msdos -C 6-1.img 1440
dd if=1.com of=6-1.img seek=18 conv=notrunc
dd if=2.com of=6-1.img seek=19 conv=notrunc
dd if=3.com of=6-1.img seek=20 conv=notrunc
dd if=4.com of=6-1.img seek=21 conv=notrunc
dd if=5.com of=6-1.img seek=22 conv=notrunc
dd if=6.com of=6-1.img seek=23 conv=notrunc
dd if=6-1.bin of=6-1.img seek=1 conv=notrunc
dd if=6-1os.bin of=6-1.img conv=notrunc

```

测试多进程并行：

对最终文件 6-1.img 的测试效果如下图：

进入内核后，输入指令 help 和 ls，可以看出 help 指令多了对 runs 指令的提示信息，ls 指令正常运行

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> help
You can input these instructions.

help      -- Print instructions
exe       -- Open an exe
exes      -- Open many exes
runs      -- run all basic exes at the same time
ls        -- Show four exes' information
cls       -- Clear the screen
exit      -- Exit OS

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> ls
exe1      -- LeftUp      512 bytes
exe2      -- RightUp     512 bytes
exe3      -- LeftDown    512 bytes
exe4      -- RightDown   512 bytes
exe5      -- RightDown   401 bytes
exe6      -- RightDown   1724 bytes

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>
```

00:15

输入指令 cls 清屏，再输入指令 exe，输入 1，调用第一个用户程序，可以看出之前的功能可以正常运行，同时可以发现，数字钟可以在用户程序运行时也同时运行

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exe
Enter a number (1-6) to open an exe:
>>
```

01:17

```
18340146 syj

  S      K Q      A      I
  T R      J L P R B Z      H J
AU Q      I OM SC Y      G K
UB P      H N N DT X      F L
W C      O G M      O U W      E M
X D      F L      F P U      U D N
Y E E MK      G      W UC O
Z FD JL H      X BT P
A CG I K I      Y A S Q
  B H      J      Z      R
```

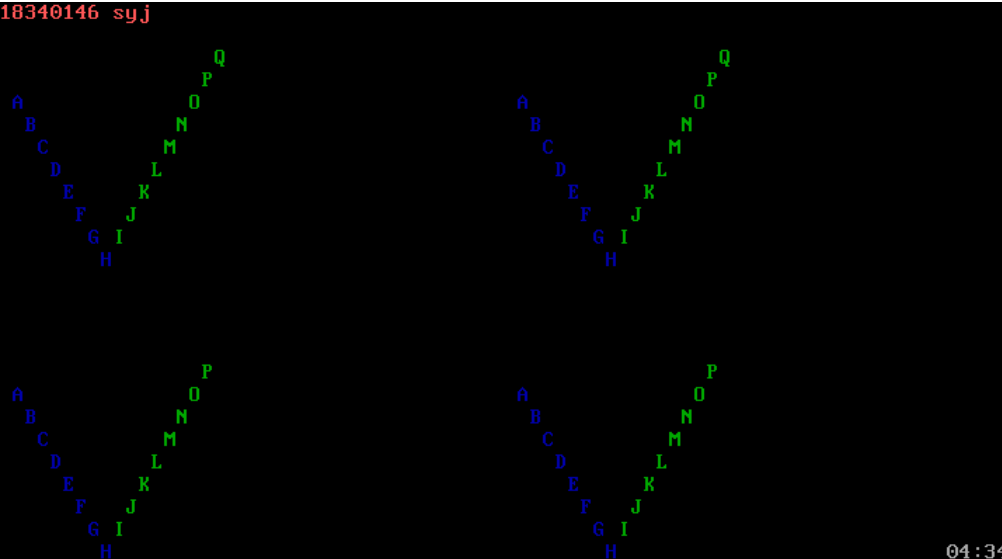
02:08

进入本次的重点测试：多进程并行。输入指令 runs，回车后可以发现四个反弹程序同时运行，但有明显的时间片轮转效果，即字符路径的显示并不完全同时同步，有时程序一会比程序三快一个字母（如下图第二张），有时程序一会比程序四快一个字母，又比程序三慢一个字母（如下图第三张）

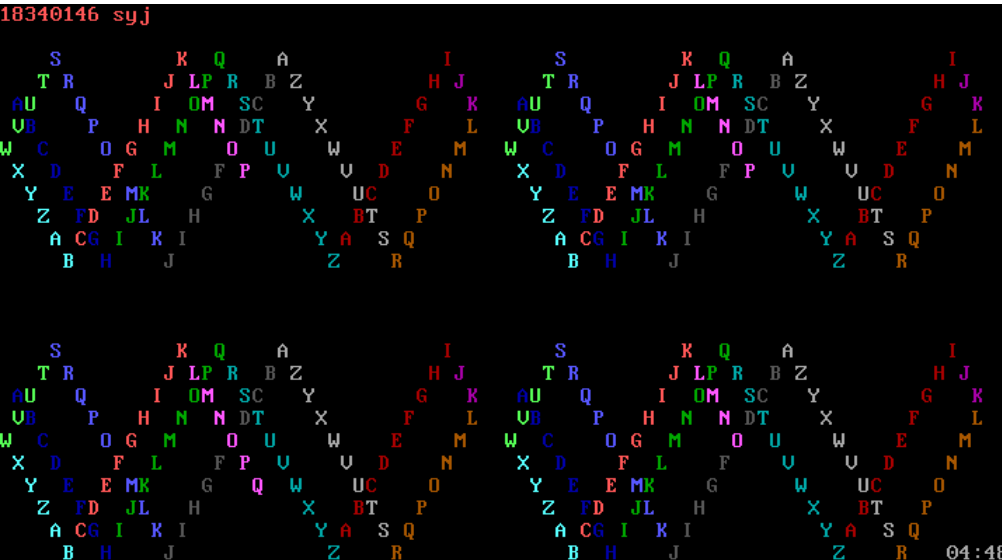
```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> runs
```



```
18340146 syj
```



```
18340146 syj
```



并行会在一段时间内自动结束，通过 `int 20h` 自动返回内核，等待下一条指令的输入

实验心得：

这次实验难度比实验五简单，任务量也少了一些，研究的新内容实际上只有进程寄存器的初始化和切换，不过研究起来也比较麻烦，寄存器一开始应该怎么赋值，是本次实验最大的难题，为此我在研究途中也走了不少的弯路

而对于其它技术层面的心得，我也在此一并讲述：

- **本次实验采用了新的调试工具 Bochs**，用于寻找进程寄存器初始化的值。在之前的实验我均未利用该调试工具，依靠理论的推导来解决寄存器和栈的问题（也没有遇到崩溃级别的问题），但是要解决本次寄存器的初始化的值，需要在内核运行到某一步时得出当时各个寄存器的值，我不得已而采用了该工具来寻找这些值，效果也比较可观
- **代码纠错**：这次程序在所有的步骤中均未参考老师代码，因此不存在老师代码纠错部分
- **实验过程中出现的出错问题及解决方式总结**：最大的难题即进程寄存器的初始化，在一开始我认为我把其中一个用户程序加载到内存 0xA300 处，那么该进程的寄存器就只需把 cs 赋值为 0x0000，ip 赋值为 0xA300 即可，但实际测试中调用 runs 指令时会卡死。我的第二个思路是：把 cs 和 ip 赋值为上述后，其他的所有寄存器赋值为与内核一样，但实际测试中调用 runs 指令时无反应，即数字钟能正常运行，但是反弹程序不显示。最后心一横，学习了新的调试工具，找到内核刚刚调用用户程序时各个寄存器的值，并手动写入 PCB 中，成为进程寄存器的初始化值，测试后发现能正常运行，就认为解决了该问题。
- **一个实际上可以避免的弯路**：在上述讲到 runs 指令卡死和无反应的问题，我的一个思考是：反弹程序使用了相同的变量名和 jmp 的位置变量名，可能是导致这个问题的原因之一。于是我强行修改了所有的相同变量名，改成不同的名字（工程量还挺大的），与此同时我也找到内核刚刚调用用户程序时各个寄存器的值，并手动写入 PCB 中（这才是解决问题的根本方式），测试后发现成功运行。我以为是修改变量名和找到正确的初始化值同时解决了这个问题，但我后来思考了一下，**用户程序是独立的，变量名实际上是地址寻值**，因此我对没有改变量名的用户程序也进行了 runs 指令，发现也能正常运行。所以改变量名是一个弯路，理论知识足够扎实是可以避免的。
- **进程的终止**：我的解决方式是修改 `int 20h`，把并行的标识变量置 0，用户程序返回到内核之后，重置 qi 指针为 PCB 数组头部，即结束并行。
- **实验5的内核其他功能，如果不必要，可暂时取消服务**：对于本次实验内容六，我的并行指令与原有的内核其他功能原理上并不冲突（因为在并行指令执行时，我暂停了内核的运行，执行结束后再恢复内核程序的运行），实际测试中也没有冲突反应，因此我无需暂时取消原有的任何服务

而整个实验下来，也是存在着一些技术上的问题并没有得到完美地解决：

- 我采取了暂停内核运行，而没有采取内核和反弹程序共同并行的方式，使得反弹程序并行时内核不能接收新的指令。这实际上应该是以后的附加实验的内容：在程序并行运行时接收键盘中断。因此在本次实验中，不实现内核与用户程序的并行。