

操作系统实验报告

18 级计科超算 18340208 张洪宾

1 实验题目

实现时间片轮转的二态进程模型

2 实验目的

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

3 实验要求

- 了解操作系统内核的二态进程模型
- 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。
- 修改时钟中断处理程序，调用时间片轮转调度算法。
- 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。
- 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。
- 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

4 实验内容

- 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。

- 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。
- 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。
- 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。
- 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。

5 实验方案

5.1 实验环境

- 操作系统：macOS Catalina 10.15.4
- 编译环境：Ubuntu 18.04 虚拟机
- 虚拟机：VirtualBox
- 汇编语言编译工具：NASM version 2.13.02
- C 语言编译工具：gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)
- 链接器：GNU ld (GNU Binutils for Ubuntu) 2.30
- 符号表工具：names(nm 命令)
- 调试工具：bochs

5.2 基本原理

- 二状态进程模型：为了实现多进程并发执行，为每个进程设置两个状态——执行和等待。进程状态由进程表保存。其中，进程状态指：进程的寄存器、序号以及状态。
- 时间片轮转：在执行多进程并发时，要在多个进程之间切换，使用的是时钟硬件中断。实现进程切换要完成以下三个步骤：保存当前进程（保存）-> 选择下一个要执行的进程（调度）-> 将要执行的进程加载入 CPU（重启进程）。

5.3 设计进程控制块

此次的进程控制块只需要在上次的寄存器表中加入进程 id 和进程状态字 state。如下：

```
1 %macro PCB 0
2     dw 0 ;ax, 偏移0
3     dw 0 ;cx, 偏移2
4     dw 0 ;dx, 偏移4
5     dw 0 ;bx, 偏移6
6     dw 0FE00h ;sp, 偏移8
7     dw 0 ;bp, 偏移10
8     dw 0 ;si, 偏移12
9     dw 0 ;di, 偏移14
10    dw 0 ;ds, 偏移16
11    dw 0 ;es, 偏移18
12    dw 0 ;fs, 偏移20
13    dw 0B800h ;gs, 偏移22
14    dw 0 ;ss, 偏移24
15    dw 0 ;ip, 偏移26
16    dw 0 ;cs, 偏移28
17    dw 512 ;flags, 偏移30
18    db 0 ;id
19    db 0 ;state
20 %endmacro
```

有了 PCB 的结构，就可以定义进程表，然后再做一步的工作了。

5.4 突破段的限制

之前的所有的用户程序，都是与内核在一个段中的，在此次实验中，我尝试在这个段内做多进程模型，结果并没有成功，上网查了很多资料，发现这种方式不利于多进程模型的实现，然后就决定修改用户程序加载到内存的位置，将它们分配到内存的不同段上。

因为 x86 的实模式下，IP 为 16 位，所以一个段的最大的长度是 2^{16} 字节，也就是 64KB。所以不考虑内核所在的段，我们应该考虑的第二个段的段基址是 10000h，并以此类推，可以得到重新分配的结果。

我们继续将这张表存在用户程序表中，然后对加载用户程序到内存中到程序做了一定的修改，重新分配的结果如下：

程序	柱面号	磁头号	起始扇区号	扇区数	功能	载入内存的位置
引导程序	0	0	1	1	打印提示信息， 加载用户信息表和内核	07C00h
用户程序表	0	0	2	2	存储用户程序存储的位置 加载到内存中的位置	7E00h
内核	0	0	3	34	常驻内存，实现 OS 最基本的功能	8000h
用户程序 1	1	0	1	2	在屏幕左上角使个人信息跳动	10000h
用户程序 1	1	0	3	2	在屏幕右上角使个人信息跳动	20000h
用户程序 3	1	0	5	2	在屏幕左下角使个人信息跳动	30000h
用户程序 4	1	0	7	2	在屏幕右下角使个人信息跳动	40000h
C 语言用户程序	1	0	9	4	输入和输出字符与字符串	0AB00h

然后我们就需要对之前加载用户程序的内核的程序做一定的修改。具体来说，就是将 int 13h 时放进 es 的 cs 改成当前段。

而在这里，由于我的设定，段基址的求法也很简单。只要加载到内存中右移四位与 0F000h 相与，则可以得到需要的段基址。而偏移量也是一样的道理：只要程序加载到内存中的地址与 0FFFFh 相与，就可以得到用户程序加载到内存中的位置在段中的偏移量了。

然后就可以通过与之前一样的方法来加载程序了。

5.5 重写 save 和 restart，并实现进程调度模块

上次我实现的 save 和 restart 模块采用的主要是宏的方式编写。而老师跟我说，如果采用宏的方式，在编译的时候会产生大量的重复代码。而这次实验主要是在进程调度的时候需要使用这几个模块，所以我主要把它放在了 wheel.asm 中，将它与时间中断写在了一起。

save 和 restart 与之前实验五的实现大同小异，我将所有寄存器都压入栈中，然后再将它们存到 PCB 中。我定义了 10 个 PCB 块，作为常量，这 10 个 PCB 块共同组成中断向量表，可以存储 10 个进程的状态（虽然本次实验我们最多用 4 个进程）。

然后我还定义了一个变量 flag 作为标志，用于标记是否进入多进程状态。然后通过它我们可以确定时间中断是否要做进程调度还是直接进入 wheel 程序。

而进程调度则是本次实验的核心，在这里我画了一个进程调度过程中栈变化的示意图，来描绘整个过程：

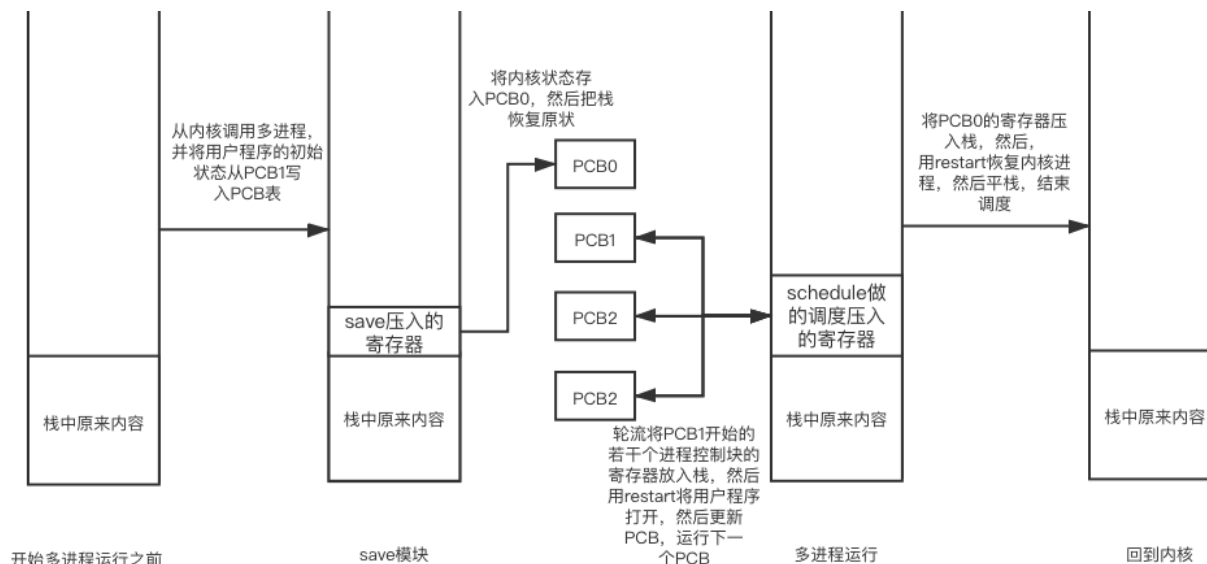


图 1: 进程调度中栈变化的示意图

然后我们就可以根据这个得到了各个模块的代码：

5.5.1 Save 模块

Save 模块和实验五的基本一致，只是在偏移量上做了一定的修改。在执行 Save 之前要将所有的寄存器压栈。

```

1 Save:
2     pusha
3     mov bp, sp
4     add bp, 16
5     mov di, all_pcb
6     mov ax, 34
7     mul word[cs:Pid]
8     add di, ax
9     ; 将栈中寄存器依次保存到进程控制块
10    mov ax, [bp]
11    mov [cs:di], ax
12    mov ax, [bp+2]
13    mov [cs:di+2], ax
14    mov ax, [bp+4]
15    mov [cs:di+4], ax
16    mov ax, [bp+6]
17    mov [cs:di+6], ax
18    mov ax, [bp+8]
19    mov [cs:di+8], ax
20    mov ax, [bp+10]

```

```

21     mov [cs:di+10], ax
22     mov ax, [bp+12]
23     mov [cs:di+12], ax
24     mov ax, [bp+14]
25     mov [cs:di+14], ax
26     mov ax, [bp+16]
27     mov [cs:di+16], ax
28     mov ax, [bp+18]
29     mov [cs:di+18], ax
30     mov ax, [bp+20]
31     mov [cs:di+20], ax
32     mov ax, [bp+22]
33     mov [cs:di+22], ax
34     mov ax, [bp+24]
35     mov [cs:di+24], ax
36     mov ax, [bp+26]
37     mov [cs:di+26], ax
38     mov ax, [bp+28]
39     mov [cs:di+28], ax
40     mov ax, [bp+30]
41     mov [cs:di+30], ax
42     popa
43
44     add sp, 16

```

5.5.2 Schedule 模块

如上图所示，Schedule 应该包含着进程的切换，不过事实上，它还应该添加一个功能，就是检测到 Ctrl + C 后，恢复内核，退出多进程用户程序。

为了配合这部分代码，用户程序需要做一定的修改，用户程序的修改在后面详细说，在这里先展示 Schedule 模块的代码：

```

1 Schedule :
2     pusha
3     mov si, all_pcb
4     mov ax, 34
5     mul word[cs:Pid]
6     add si, ax ; si 指向进程控制块表的地址，即第一个进程控制块的地址
7     mov byte[cs:si+33], 1
8
9     mov ah, 01h
10    int 16h
11    jz NextPCB
12    mov ah, 00h
13    int 16h

```

```

14     cmp ax, 2e03h          ; 检测 Ctrl + C
15     jne NextPCB
16
17     mov word[cs:Pid], 0
18     mov word[cs:pflag], 0    ; 将并行标志设置为不允许并行
19     call resetAllPcbExceptZero
20     jmp QuitSchedule

```

5.5.3 Restart 模块

Restart 模块和实验五的也差不多，在这里列一下代码：

```

1 Restart:
2     mov si, all_pcb
3     mov ax, 34
4     mul word[cs:Pid]
5     add si, ax              ; si 指向调度后的PCB的首地址
6
7     mov ax, [cs:si+0]
8     mov cx, [cs:si+2]
9     mov dx, [cs:si+4]
10    mov bx, [cs:si+6]
11    mov sp, [cs:si+8]
12    mov bp, [cs:si+10]
13    mov di, [cs:si+14]
14    mov ds, [cs:si+16]
15    mov es, [cs:si+18]
16    mov fs, [cs:si+20]
17    mov gs, [cs:si+22]
18    mov ss, [cs:si+24]
19    add sp, 11*2            ; 恢复正确的 sp
20
21    push word[cs:si+30]      ; 新进程 flags
22    push word[cs:si+28]      ; 新进程 cs
23    push word[cs:si+26]      ; 新进程 ip
24
25    push word[cs:si+12]
26    pop si                  ; 恢复 si
27 QuitParallel:
28    push ax
29    mov al, 20h
30    out 20h, al
31    out 0A0h, al
32    pop ax
33    iret

```

5.6 将用户程序初始状态加载到进程控制块中

我们在多进程执行用户程序之前，需要将所有要运行的用户程序加载到内存并初始化，并放入从 PCB1 开始的进程控制块中。我们可以根据 shell 的输入，确定要运行哪几个程序，然后将它们加载到 PCB 中，然后轮流执行。

具体的代码如下：

```
1 loadProcessMem:                                ;函数： 将某个用户程序加载入内存并初始化其PCB
2     pusha
3     mov bp, sp
4     add bp, 16+4                                ;参数地址
5     LOAD_TO_MEM [bp+12], [bp], [bp+4], [bp+8], [bp+16], [bp+20]
6
7     mov si, all_pcb
8     mov ax, 34
9     mul word[bp+24]                             ;progid_to_run
10    add si, ax                                  ;si 指向新进程的PCB
11
12    mov ax, [bp+24]                             ;ax=progid_to_run
13    mov byte[cs:si+32], al                      ;id
14    mov ax, [bp+16]                             ;ax=用户程序的段值
15    mov word[cs:si+16], ax                      ;ds
16    mov word[cs:si+18], ax                      ;es
17    mov word[cs:si+20], ax                      ;fs
18    mov word[cs:si+24], ax                      ;ss
19    mov word[cs:si+28], ax                      ;cs
20    mov byte[cs:si+33], 1                      ;state 设其状态为就绪态
21    popa
22    retf
```

5.7 在多进程状态下关闭风火轮，其余状态下显示风火轮

最开始我没有考虑到这点，所以当进入多进程状态的时候，屏幕会被卡住。我通过 bochs 调试之后，发现风火轮程序运行的时候破坏了原来的栈，这样使得整个进程调度模块出现了问题。

然后我就决定，如果在进入多进程状态的时候，将 flag 置为 1，然后关闭风火轮，否则，打开风火轮。

5.8 修改用户程序

进入用户程序比较简单，但是在多进程状态下，退出用户程序，就不能用原来在各个用户程序中都使用 Ctrl + C 的方式了，因为这样一次只能结束一个用户程序，而且还是随机结束的。但是我们也

不能取消掉用户程序中的 Ctrl + C，因为我们将程序进行批处理运行的时候，如果没有这个退出的机

制，我们就会无法退出用户程序。

然后我就开始思考：我们需要在用户程序中判断当前状态是多进程状态还是批处理状态。这两种状态的差别在于内核中的 flag 的值的不同。而由于 flag 在内核中，无法共享，我决定使用系统调用来获取这个值。

通过这个系统调用，我获取了内核中 flag 的值放到 ax 中，然后根据它来判断是否要用 Ctrl + C 退出整个用户程序。

5.9 文件结构

与上次相比，我并没有改变文件结构，所以 Makefile 没有发生变化。

5.10 多进程运行接口

在原来的基础上，我增加了一条新的命令：parallel，用于多进程运行用户程序。

当 shell 输入 parallel 命令的时候，就需要运行多进程的函数。多进程函数如下：

```
1 void parallel_run(char*user_input){
2     clean();
3     char arguments[MaxSize] = {0};
4     unsigned short all_id[MaxSize] = {0};
5     int count = 0;
6     int flag = 1;
7     for(int i = 0; i < MaxSize; i++){
8         get_argument(user_input, arguments);
9         char temp[16] = {0};
10        get_i_th_argument(arguments, temp, i + 1);
11        if(temp[0] == 0){
12            break;
13        }
14        if(is_num(temp) == 0){
15            flag = 0;
16            break;
17        }
18        all_id[count++] = string_to_num(temp);
19    }
20    unsigned short total = get_number_of_program();
21    for(int i = 0; i < count; i++){
22        if(all_id[i] > 4){
23            flag = 0;
24        }
25    }
26    if(flag){
27        int i = 0;
28        for(int i = 0; i < count; i++){
```

```

29     int progid_to_run = all_id[i]; // 要运行的用户程序 Progid
30     loadProcessMem(get_cylinder(all_id[i]), get_head(all_id[i]),
31     get_sector(all_id[i]), get_size(all_id[i])/512,
32     get_segment(all_id[i]), get_address(all_id[i]), progid_to_run);
33 }
34 pflag = 1; // 允许时钟中断处理多进程
35 Delay();
36 pflag = 0; // 禁止时钟中断处理多进程
37 clean();
38 }
39 else{// 参数无效
40     char* error = "Invalid arguments.\r\n";
41     print(error);
42 }
43 }

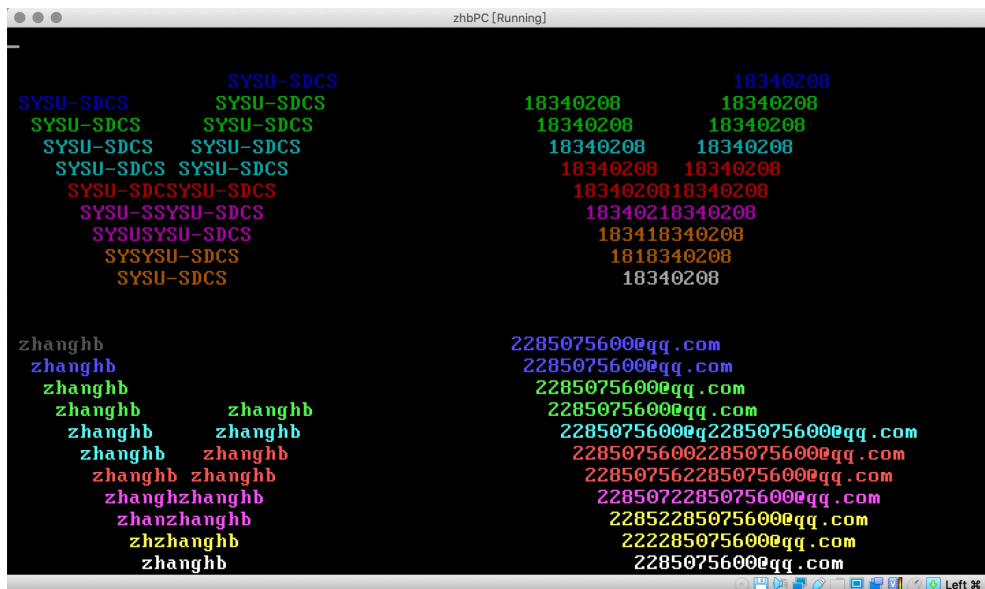
```

在内核中调用该函数，然后就可以调用多进程模型了。

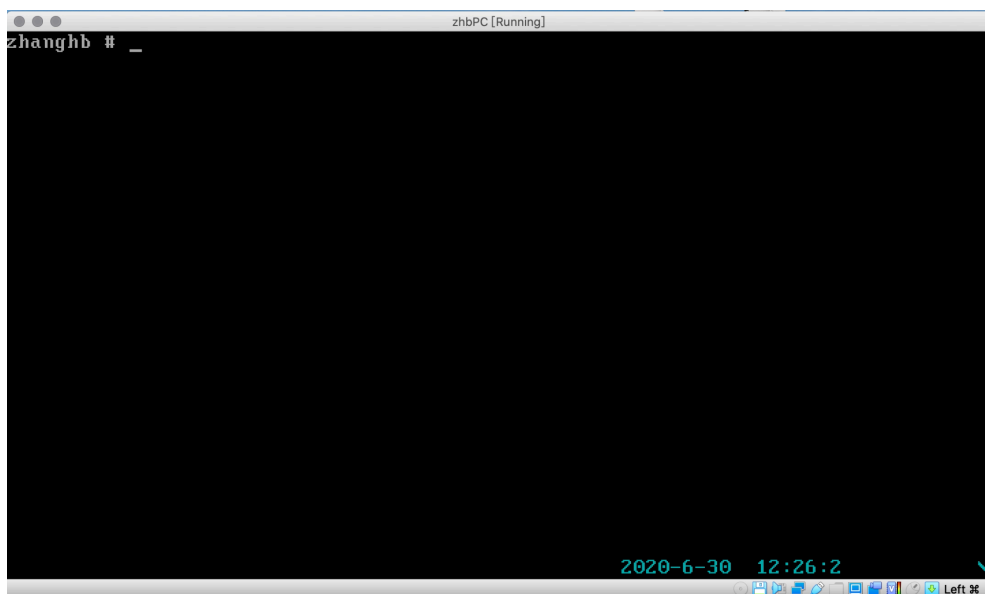
6 实验结果

在附近中的视频显示了多进程的示意图，但是由于进程切换的速度不够快，无法用截图的方式将进程切换体现出来，于是我又临时将用户程序中清除上次打印内容的程序删除了，得到了多进程的效果图。

在终端中输入 parallel 1 2 3 4，按下回车，如下：（在这里随机截取了两个状态）



可以看到，四个进程同时运行。按下 Ctrl + C，退出执行，如下：



可以看到程序正确返回命令行。

7 实验总结

这次实验总体上来说思路比较清晰，但是过程非常艰巨。

原理非常简单，这是整个操作系统进程模型中最简单的一个。但是实现起来仍然花费了我大量的时间。

7.1 遇到的问题

- 在给用户程序分段的过程中，我最开始没有意识到要将加载到内核中的地址右移才能求段值，所以最开始的用户程序根本无法正确加载，在这里我找了很久的 bug
- 在做进程调度的过程中，栈的处理依旧让我苦恼，因为栈的一点点误差导致结果无法运行，也让我花费了大量的时间来调试。

7.2 心得体会

通过这次实验，我的汇编编程技巧得到了较大的提升，然后调试的技巧也逐渐提升，学会用 bochs 调试时间中断，并最终解决了问题。

虽然实验非常困难，但是当程序正确运行的时候还是非常激动的。通过这次实验也大大提升了编程的技巧，收获颇丰。