进程模型

- 二状态的进程模型
- 进程控制块与进程表
- 进程交替执行原理
- 利用时钟中断实现用户程序轮流执行
- 参考原型
- 实验项目6说明

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

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

显示区划分

1 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
  - 8086CPU的所有寄存器：AX/BX/CX/DX/BP/SP/DI/SI/CS/DS/ES/SS/IP/FLAG
  - 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
- 汇编语言中描述PCB
- C语言描述

汇编语言中描述PCB

```
PCB:
        dw 0B800h; GS \
        dw 6000h  ; FS   | 部分段寄存器，用PUSH指令一个个压入栈
        dw 6000h  ; ES   |
        dw 6000h  ; DS /
        dw 0      ; DI   \
        dw 0      ; SI    | 指针寄存器 \
        dw 0      ; BP    |              |
        dw 100h-4; SP /                  | 通用寄存器，用PUSHA指令一起压入栈
        dw 0      ; BX  \                |
        dw 0      ; DX   | 主寄存器     /
        dw 0      ; CX   |
        dw 0      ; AX  /
        dw 6000h ; SS 堆栈段寄存器，手工赋值
        dw 100h  ; IP 指令指针寄存器                    \
        dw 6000h ; CS 代码段寄存器                       | 中断时由CPU压入栈
        dw 512   ; Flags 标志寄存器(中断允许IF＝1)  /
        dw 1      ; ID 进程ID
        db 'ProcessA' ; Name 进程名（8个字符）
```

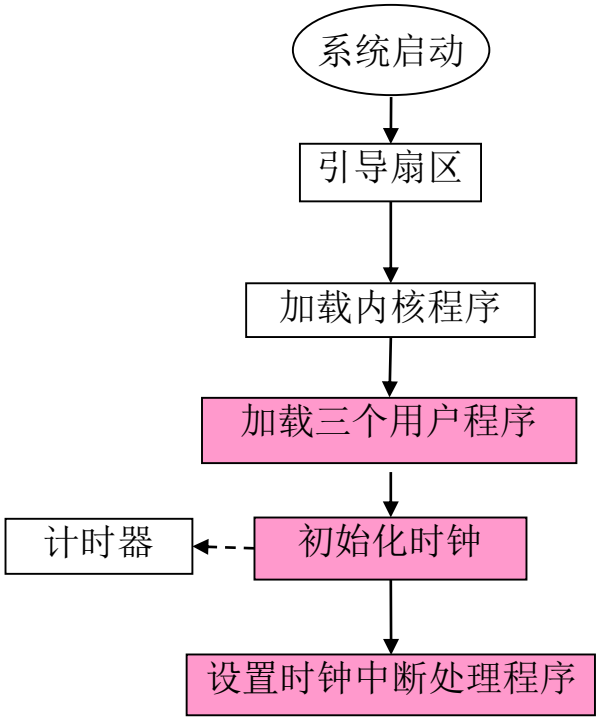
汇编语言中描述PCB

```
typedef struct RegisterImage{
        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;
}RegisterImage;
typedef struct PCB{
        RegisterImage regImg;/***registers will be saved in this struct automactically by timer interrupt***/
        int ID;
      PCB *fPCB
        PCB_Status status;
}PCB;
```

2进程交替执行原理

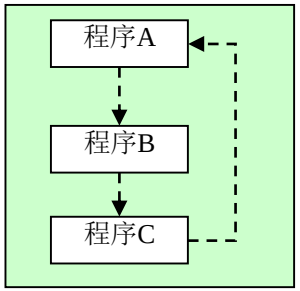
- 在以前的原型操作系统顺序执行用户程序，内存中不会同时有两个用户程序，所以CPU控制权交接问题简单，操作系统加载了一个用户到内存中，然后将控制权交接给用户程序，用户程序执行完再将控制权交接回操作系统，一次性完成用户程序的执行过程
- 采用时钟中断打断执行中的用户程序实现CPU在进程之间交替
- 简单起见，我们让两个用户的程序均匀地推进，就可以在每次时钟中断处理时，将CPU控制权从当前用户程序交接给另一个用户程序

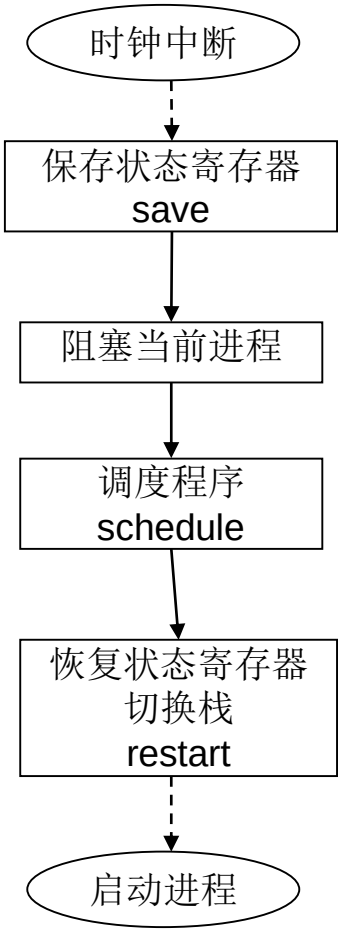
实现进程模型 的系统框架



全局逻辑图

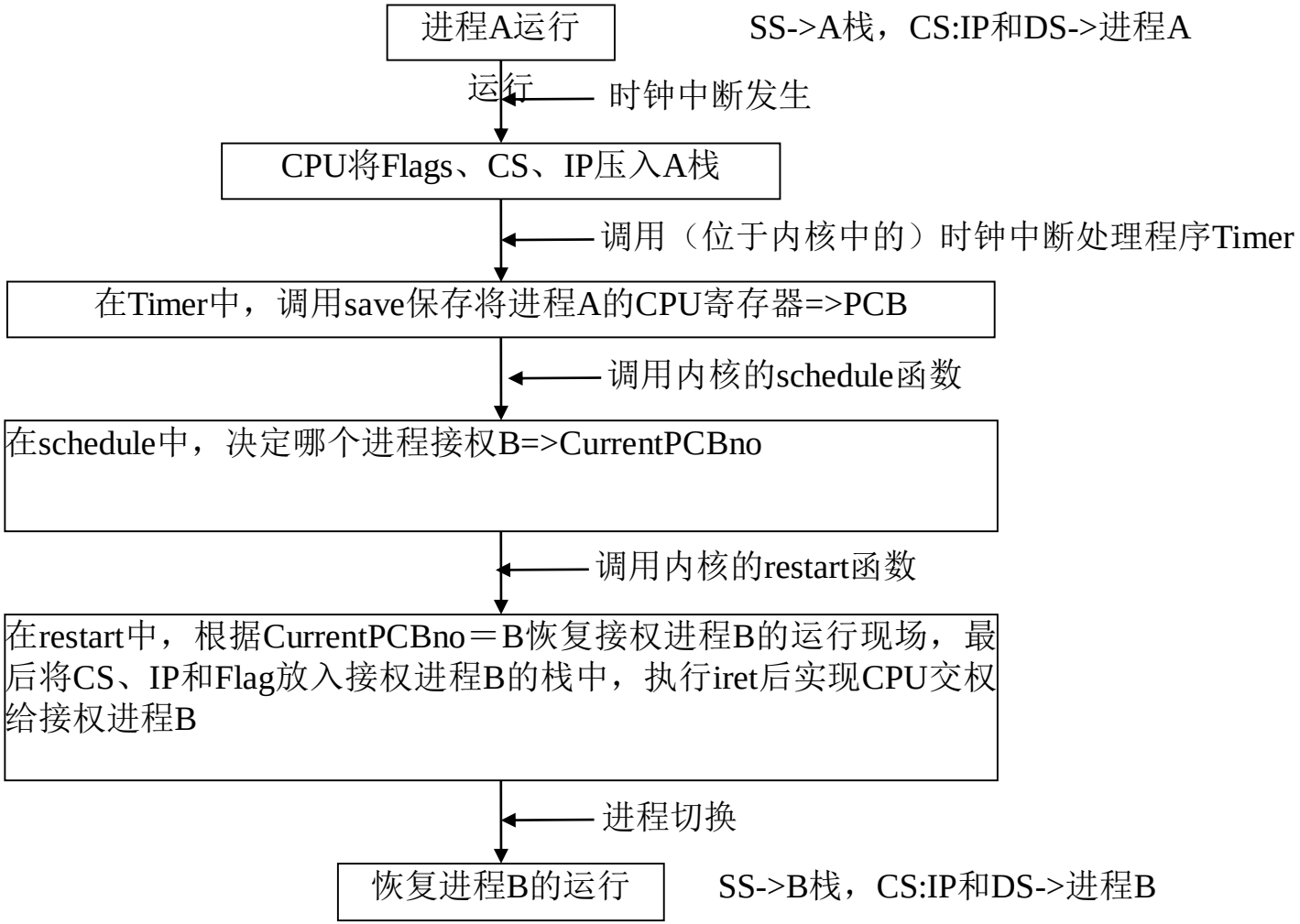
（红色为内核模块）





3 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。


进程A运行
SS->A栈，CS:IP和DS->进程A
运行
时钟中断发生
CPU将Flags、CS、IP压入A栈
调用（位于内核中的）时钟中断处理程序Timer
在Timer中，调用save保存将进程A的CPU寄存器=>PCB
调用内核的schedule函数
在schedule中，决定哪个进程接权B=>CurrentPCBno
调用内核的restart函数
在restart中，根据CurrentPCBno＝B恢复接权进程B的运行现场，最后将CS、IP和Flag放入接权进程B的栈中，执行iret后实现CPU交权给接权进程B
进程切换
恢复进程B的运行
SS->B栈，CS:IP和DS->进程B

现场保护: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。
- 涉及到三种不同的栈：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。

现场保护:save过程

```
;*************************************************************************
;*                    Save                                               *
;*************************************************************************
; 将当前进程的寄存器值保存到当前进程的PCB中
Save:                   ;当前栈顶：*\psw\cs\ip\call_ret
    push ds             ;当前栈顶：*\psw\cs\ip\call_ret\ds(用户)
    push  cs            ;当前栈顶：*\psw\cs\ip\call_ret\ds(用户)\cs(内核)
    pop  ds             ;当前栈顶：*\psw\cs\ip\call_ret\ds(用户)
                        ;当前ds指向内核
    pop word ptr ds:[ds_save]    ;当前栈顶：*\psw\cs\ip\call_ret
    pop word ptr ds:[ret_save]   ;当前栈顶：*\psw\cs\ip
    mov word ptr ds:[bx_save],bx
    mov word ptr ds:[ax_save],ax
    mov bx,word ptr ds:[p_proc_ready]
    mov word ptr ds:[(bx+SS_OFF)],ss  ;PCB[*]:SS
    pop word ptr ds:[(bx+IP_OFF)]       ;当前栈顶：*\psw\cs  ---->;PCB[*]:PSW(X)\CS(X)\IP\SS
    pop word ptr ds:[bx+CS_OFF)]        ;当前栈顶：*\psw ;PCB[*]:PSW(X)\CS\IP\ss
    pop word ptr ds:[(bx+FLAG_OFF)]   ;当前栈顶：*;PCB[*]:PSW\CS\IP\ss
    mov word ptr ds:[sp_save],sp
    add bx,AX_OFF+2
    mov ax,bx
    mov sp,ax
    push cs
    pop ss
```

现场保护:save过程

```
push word ptr ds:[ax_save]      ;PCB[*]:PSW\CS\IP\SS\ax
push cx                         ;PCB[*]:PSW\CS\IP\SS\ax\cx
push dx                         ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx
push word ptr ds:[bx_save]      ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx
push word ptr ds:[sp_save]      ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx\sp
push word ptr ds:[bp_save]      ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp\bp
push si                         ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx\sp(X)\bp\si
push di                         ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp(X)\bp\si\di
push word ptr ds:[ds_save]      ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp(X)\bp\si\di\ds
push es;                        ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp(X)\bp\si\di\ds\es
push fs                         ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp(X)\bp\si\di\ds\es\fs
push gs;                        ;PCB[*]:PSW\CS\IP\SS\ax\cx\dx\bx(X)\sp(X)\bp\si\di\ds\es\fs\gs
mov sp,word ptr ds:[kernal_sp]          ;中断处理期间使用内核的栈
mov ss,word ptr ds:[kernal_ss]
mov bx,ds
mov es,bx                       ;中断处理期间es指向的数据段
mov ax,word ptr ds:[ret_save]
jmp (ax)
```

进程切换: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。

现场恢复:restart过程

```
;*************************************************************************
;*                    Restart                                            *
;*************************************************************************
; 从当前选择的进程的PCB中恢复寄存器值，并启动其运行
Restart:
    mov ss,word ptr cs:[kernal_cs]
    mov sp,word ptr cs:[p_proc_ready]   ;以当前进程的PCB为临时栈，恢复寄存器值
    pop gs
    pop fs
    pop es
    pop ds
    popa
    pop ss                                  ;恢复新进程的栈段SS     ;进程栈：
    mov word ptr cs:[bx_save],bx
    mov bx,word ptr cs:[p_proc_ready]
    push word ptr cs:[(bx+FLAG_OFF)]    ;当前新进程栈：*\psw
    push word ptr cs:[(bx+CS_OFF)]      ;当前新进程栈：*\psw\cs
    push word ptr cs:[(bx+IP_OFF)]      ;当前新进程栈：*\psw\cs\ip
    mov bx, word ptr cs:[bx_save]
    sti
    iret                                            ; 切换到新进程
```

时钟与中断处理

```
;设置计时器，时钟每秒20次中断（50ms一次）：
;*************************************************************************
;*                    SetTimer                                           *
;*************************************************************************
; 设置计时器函数
SetTimer:
    mov al,34h                          ; 设控制字值
    out 43h,al                                  ; 写控制字到控制字寄存器
    mov ax,1193182/20          ; 每秒20次中断（50ms一次）
    out 40h,al                                  ; 写计数器0的低字节
    mov al,ah                                   ; AL=AH
    out 40h,al                                  ; 写计数器0的高字节
    ret
; ---------------------------------------------------------
时钟中断处理程序参考程序：
;*************************************************************************
;*                    Timer                                              *
;*************************************************************************
; 时钟中断处理程序
Timer:              ; 使用当前进程的栈
    call Save           ;当前进程的栈增加ret_addr
    ret_addr: call Scheduler
    call Restart
```

进程调度：Scheduler过程

```
;汇编程序实现的进程调度函数，采用时间片轮转法。
;*************************************************************************
;*                    Scheduler                                          *
;*************************************************************************
; ---------------------------------------------------------
; 进程调度函数：时间片轮转
Scheduler: ;

    ; 计算下一个PCB地址，变量p_proc_ready含当前进程PCB起始地址
    add word ptr [p_proc_ready],PCB_Size    ; p_proc_ready = 下一进程PCB的起始地址
    cmp word ptr [p_proc_ready],MaxAddr     ; 比较p_proc_ready与进程表上限值
    jl Less                                 ; < 上限值：跳转到Less处
    mov word ptr [p_proc_ready],PCB1        ; >= 上限值：p_proc_ready = PCB1
Less:
    ret
; ---------------------------------------------------------
```

6 实验项目6

- 保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：
- (1)在c程序中定义进程表，进程数量为4个。
- (2)内核一次性加载4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。
- (3)在原型中保证原有的系统调用服务可用。再编写1个用户程序，展示系统调用服务还能工作。

5 参考原型

- BOOT
- MyOS.asm主程序
- kLiba.asm内核库
- cMain.c程序
- Kernal.h头文件
- kLibc.asm库