实时多任务系统内核分析

初次接触实时多任务操作系统的人，往往对实时程序的运行机制感到很困惑：任务在什么时候投入运行？操作系统以什么机制决定目前应该运行哪一个任务？本任务什么时候放弃了对 CPU 的控制？为了解答以上问题，我们从分析一个很简单的实时调度程序入手，来说明一下多任务程序的运行过程。

从结构上来说，实时多任务操作系统包括两部分，一部分为操作系统内核(kernel)，即实时执行程序(Real Time Executive:RTX)，另一部分是输入输出部分(I/O)(注意开发系统不属于操作系统的范畴)；嵌入式系统对 I/O 的需求通常比较小(无文件系统需求)，因此很多实时多任务操作系统本质上就是一个实时执行程序，如 AMX(Kadak)，VRTX(Microtec)，iRMX(Intel)等(这里的 X 即:eXecutive)，如果纯粹从 kernel 的角度来考察目前流行的各种实时多任务操作系统的性能，它们的效率差别都不大。

在市面上可以得到一些 RTX 的源代码(有用 C 实现的，有用汇编实现的，还有用 PL/M 语言实现的)，从 internet 上也可以荡一些下来(我介绍一个站点www.eg3.com，堪称世界电子工程师资源宝库)，下面我要介绍的一个 RTX 版本(我命名为 SRTX:short RTX)，可以说是 RTX 中的元老级产品了，来自某研究所，九十年代初他们到美国考察，从美国某公司购得。五年以前，SRTX 在国内有售，许多搞工控的研究所利用 SRTX 开发了一些大型或小型的产品,这里介绍的 SRTX 我作了一些简化和改动.

在功能上，SRTX 的确无法和目前市售的实时多任务操作系统相比，不支持任务的调试，不支持优先级反转，甚至不支持相同优先级任务的分时间片运行；SRTX 的功能单一，程序代码非常短，效率高(毕竟是 80 年代末的产品)。从内核的角度看，SRTX 实现了一个较基本的任务调度版本；因此通过对 SRTX 的介绍，可以了解其他实时多任务产品内核的结构及实现方法。

一. 任务的管理及程序实现

1. 任务及程序结构：

从程序实现上说，任务就是一段能完成既定功能的程序代码。与一般的程序代码（或子程序）不一样的地方就在于任务是死循环的程序结构。

任务的通用程序结构如下：

```
#define void TASK
TASK Common_Task( void )
{
     Task_ Init(); // initialize data structure for this task
     while( 1 )    // loop forever
     {
          Suspend_Task_for_Msg(); //wait for message
          Process_This_Msg();     //process this message
          Post_Msg_to_Task();     //send message to other task
     }
}
```

举例说明：

应用户的要求，需要在屏幕的右上角显示当前时钟，我们可以把此功能当作一个专用的任务来设计，该任务的功能是每隔一秒取出系统当前的时钟，转换成规定格式的字符串，将其指针传送给显示任务，接着继续等下一秒的到来。任务的程序实现结构为：

```
TASK  Alert_Clock( void )
{
     Set_String_NULL();                    // initialize string
     while( 1 )
     {
          Get_Current_Clock( );             //get clock
          Format_Clock_Str( );              //format digit to string
          Send_Msg_To_Task( DISPLAY,str ); //post message pointer to
                                            //DISPLAY task
          Suspend_Task( 100 );              //suspend 1 second
     }
}
```

2. 数据区及堆栈区的组织方法

以下讲一下 SRTX 的系统数据区及堆栈区的组织方法。

任务在运行的时候可能被更高优先级的任务中断，这时候任务需要将现场信息放到其堆栈中，以便今后能从该堆栈中取出被中断的现场信息（包括断点位置，任务状态等）恢复任务的断点运行，任务需要有自己独立的堆栈区以及描述任务运行状态措施，在实时多任务系统中，采用一种任务控制块（Task Control Block:TCB）的数据结构描述任务的运行状态，每个任务有一个 TCB。

在 SRTX 中，采用如下方法定义任务的堆栈：

```
max_task_no     equ     32              ;support maxinum tasks
                                        ;defined by myself
```

```
;---------------------------------
;stacks for all tasks
;---------------------------------
stack   segment para stack 'stack'
        db      max_task_no*1024 dup(?)
stack   ends
```

每个任务的堆栈大小为 1K，用户可以根据需要修改(同时需要改动程序)。

在 SRTX 中，TCB 的结构为：(共 16 个字节:10h)

表 1 任务的 TCB 的结构

偏移（字节）	说 明
0	任务号:Task_ID
1	任务状态:Task_Status
2—3	初始代码偏移:Init_Offset
4—5	初始 CS 段值:Init_CS
6—7	初始堆栈指针 SP:Init_SP
8—9	初始堆栈段 SS:Init_SS
10—11	上下文堆栈指针 SP:Last_SP
12—13	上下文堆栈段 SS:Last_SS
14—15	系统时钟援用:Slice

以下是系统数据区的组织方法：

```
;---------------------------------------
;buffer for task and queue manage
;---------------------------------------
dseg    segment
        db      10h     dup(?)          ;00h-> RUNning task message
                                        ;      public for all tasks
        db      "taskdescripitors"      ;10h-> TCB starting flag
        db      max_task_no*10h dup(?)  ;20h-> task TCBs message
        db      "queue tables    "      ;220h->QUEUE starting flag
        db      max_task_no*108h dup(?) ;230h->all queues message
        db      "SRTX version 1.0"      ;      SRTX version flag
dseg    ends
```

其中："taskdescripitors"为 16 字节的标志串，以标明 TCB 的开始；

"queue tables   "为 16 字节的标志串，标明任务队列的开始；

"SRTX version 1.0"为 16 字节的版本标志，

关于 TCB 各字段的含义以及 SRTX 的队列组织方法见相应章节。

3． 任务状态及其表示

在 TCB 的 Task_Status 字段用于描述本任务当前的状态，TCB 状态及其它信息也在本字段中描述。

表 2 任务状态及其描述

位	状 态	描 述
10000000	INHIBIT	任务被强制禁止执行
00100000	AVAILABLE	本 TCB 为空，可以使用
00010000	INTERRUPT	任务运行被中断
00001000	ASLEEP	任务处于睡眠状态(调用 SYSSLP)
00000010	AWAKENED	任务睡眠超时,被唤醒
00000001	RUNNING	任务处于可运行状态(ready to run)

以上描述了系统数据区的组织方法,尤其描述了其中的任务状态字节,以下程序代码用于系统运行初期对系统数据区的初始化,它是最先需要运行的代码.

```
;-----------------------------------------
        mov     ax,dseg
        mov     ds,ax
        cld
        push    ds
        pop     es
        xor     di,di                   ; initialize public area
        mov     cx,10h
        xor     ax,ax
        repz    stosb                   ; set 0 for all 10h bytes

        mov     bx,20h                  ; bx->first tcb
        mov     cx,max_task_no
task_dec:
        mov     byte ptr[bx+01],20h     ; initialize tcb
        add     bx,10h                  ; set TCBs are all AVAILABLE
        loop    task_dec

         mov     bx,0230h
         mov     cx,max_task_no
 task_queue:
         mov     byte ptr[bx+0100h],80h  ; initialize queue is empty
         add     bx,0108h
         loop    task_queue
```

4. 任务创建及启动

为了详细说明 TCB 中一些字段的使用，我们介绍在 SRTX 中的创建任务例程。

在 SRTX 中，syscre 为任务创建例程，其使用方法是：

输入：寄存器 DX：AX 包含初始任务代码段的 CS：offset

输出：若创建成功，CARRY 清 0；此时，AX 包含了任务的 ID 号，它是唯一的任务标识符。若 CARRY 置位，AX 寄存器将包含下列错误码：03eah……任务创建错误。

以下是 SYSCRE 的典型使用方法：

```
;-------------------------------------------------------
       mov  dx, seg    NEW_TASK  ;put seg to dx
       mov  ax, offset NEW_TASK  ;offset to ax
       call SYSCRE               ;creat NEW_TASK
       jc   creat_error          ;creat error process
       mov  NEW_TASKID,ax        ;put task id to NEW_TASKID
       call SYSSTR               ;startup this task
;-------------------------------------------------------
```

以下是具体的实现代码:

```
;--------------------------------
;syscre: create a task
;--------------------------------
syscre_sub:
        push    bx
        push    cx
        push    si
        push    ds                      ;reserve used regs
        mov     bx,dseg
        mov     ds,bx

        mov     si,0                    ;first task ID=0
        mov     bx,20h                  ;bx->first TCB
        mov     cx,max_task_no          ;max loop count

find_empty_tcb:
        test    byte ptr[bx+01],20h     ; is this tcb empty?
        jnz     fill_task_msg           ; yes,TCB is AVAILABLE
                                        ; please see table 2
        add     bx,10h                  ; next tcb
        inc     si                      ; task ID++
        loop    find_empty_tcb

        mov     ax,03eah                ; no tcb AVAILABLE
        jmp     syscre_error

fill_task_msg:
        mov     cl,0                    ; initial task status
```

```
        mov     [bx+01],cl              ; save initial task status
        mov     [bx+02],ax              ; offset of task code
        mov     [bx+04],dx              ; segment of task code
        mov     ax,si                   ; si:task ID
        mov     cx,0400h                ; stack length
        mul     cx
        add     ax,0400h                ; initial task stack pointer
                                        ; stack bottom =SP
        mov     [bx+06],ax              ; set initial sp
        mov     word ptr [bx+08],stack  ; set initial stack segment
        mov     ax,si                   ; task's ID
        mov     [bx+0],al               ; reserve task ID
        clc
        jmp     syscre_exit

syscre_error:
        stc

syscre_exit:
        pop     ds                      ;restore used regs
        pop     si
        pop     cx
        pop     bx
        ret
```

由此可见，在 TCB 中,Task_ID 表示任务创建的顺序号(系统创建的第一个任务,Task_ID=0,创建的第二个任务，Task_ID=1……)，任务创建后，Task_Status＝0；所有的状态清除。Init_Offset=任务代码的 offset,Init_CS=任务代码的 CS，Init_SP=该任务创建时系统分配给该任务的 SP,Init_SS=该任务创建时系统分配给该任务的 SS,在任务创建阶段,TCB 的其它域未涉及到。

在 SRTX 中，sysstr 为任务启动例程，其使用方法是：

输入：寄存器 AX 包含待启动任务的 ID 号;

输出：若启动成功，CARRY 清 0；否则，CARRY 置位，AX 寄存器将包含下列错误码：

03e9h……无效任务 ID 号。

关于 SYSSTR 的使用示例见上.

以下是 SYSSTR 的实现代码.

```
;-------------------------------
;sysstr start(or restart task)
;-------------------------------
sysstr_sub:
        push    ax
        push    bx
        push    cx
        push    ds                      ; reserve used regs
```

```
        mov     bx,dseg
        mov     ds,bx
        cmp     ax,max_task_no
        jnb     sysstr_error            ;task ID is too large

        mov     bx,20h
        mov     cl,04                   ; size of TCB=16,by <<4
        shl     ax,cl                   ; get TCB addr for this task
        add     bx,ax                   ; bx->this task's TCB
        test    byte ptr[bx+01],20h     ; task created ?
        jnz     sysstr_error            ; task is not created still
        or      byte ptr[bx+01],01      ; set runnable flag
        clc
        jmp     sysstr_exit

sysstr_error:
        mov     ax,03e9h
        stc

sysstr_exit:
        pop     ds                      ; restore used regs
        pop     cx
        pop     bx
        pop     ax
        ret
```

在 SRTX 中,启动一个任务实质上就是设置该任务的状态为可运行状态,等待调度程序 reschedul 运行时根据优先级启动该任务的运行.

5. 任务删除及其实现方法

SYSDEL 用于删除一个已经创建的任务,释放该任务占用的 TCB.

SYSDEL 的使用方法:

输入:AX 包含将要删除任务的标识符.

输出:如果删除成功,CARRY 清零,否则,CARRY 置位,AX 寄存器包含如下的错误码…… 03e9h,无效任务 ID 号。

以下是 SYSDEL 的实现代码。

```
;-----------------------------------------
;       sysdel delete a task
;-----------------------------------------
```

```
sysdel_sub:
        push    bx
        push    ds
        mov     bx,dseg
        mov     ds,bx
        cmp     ax,max_task_no
        jnb     sysdel_sub_error        ;task id is too large
        mov     bx,10h
        mul     bx
        mov     bx,20h
        add     bx,ax                   ;bx->tcb for this task ID
        mov     al,[bx+01]
        or      al,20h                  ;tcb is now AVAILABLE
        mov     [bx+01],al
        clc
        jmp     sysdel_sub_exit
sysdel_sub_error:
        mov     ax,03e9h
        stc
sysdel_sub_exit:
        pop     ds
        pop     bx
        ret
```

任务在创建之后,该任务的 TCB 的 AVAILABLE 字段被设置为 0,表明该任务对应的 TCB 不可用(或已经被占用);SYSDEL 实质上就是简单地将该任务的 TCB 的 AVAILABLE 字段被设置为 1,重新置为可用(未被占用)，这样，今后在创建一个新任务时，可以重新利用该 TCB。

6. 禁止任务切换、允许任务切换的实现

在需要对临界区数据进行访问的时候,需要禁止任务的切换,以保护共享的资源.

举例如下:

任务 A 需要任务 B 的数据以便继续操作,任务 A 和任务 B 通过设置公共变量 READY(信号量)来表示数据是否准备好,任务 B 设置 READY 的初值为 false,一旦 B 将数据放入公共的缓冲区之后,设置 READY 为 true;任务 A 挂在信号量 READY 上,若 READY 为 true,任务 A 从公共的缓冲区取出 B 存放的数据.对公共数据的访问就成了临界区的操作.这主要是由于,在 A 测试 READY 为 true 之后,马上取公共缓冲区的数据,如果这时候发生任务的切换,任务 B 开始运行,它可能将最新的数据放入公共缓冲区中,这样任务 A 恢复

运行后,将继续读取公共缓冲区的数据,使得任务 A 读取的数据不完整,达不到预期的效果.

对临界区资源的访问时,一定要禁止任务的切换,以保护资源的完整性.

SRTX 采用了最简单的关门算法实现对任务切换的管理,即:

需要对临界资源进行访问时,任务 A 首先将某全局变量 M 减 1,然后访问临界资源;操作系统的调度程序 reschedul 首先判断 M 是否等于 0,如果 M 不等于 0,则不能进行任务的切换(即不能让高优先级的任务打断),继续运行当前的任务,直到完成; 等于 0,表明任务已经不需要该资源,可以进行任务的切换.任务 A 对临界资源访问完毕后,将 M 加 1,实现配对操作.

以下是任务禁止切换和任务允许切换的实现代码:

```
;-------------------------------------------
;sysdti-disable task interrpt(task switching)
;-------------------------------------------
sysdti_sub:
        pushf
        push    ds
        push    ax
        mov     ax,dseg
        mov     ds,ax
        dec     ds:byte ptr[07]         ; disable flag
        pop     ax
        pop     ds
        popf
        ret
;-------------------------------------------
;syseti:enable task interrpt( task switching)
;-------------------------------------------
syseti_sub:
        pushf
        push    ds
        push    ax
        mov     ax,dseg
        mov     ds,ax
        inc     ds:byte ptr[07]         ; enable flag
        pop     ax
        pop     ds
        popf
        ret
```

这里可以看出系统数据区公共字段中的第 7 字节的用途,当第 7 字节不等于 0 时,

不允许发生任务的切换(系统数据区公共字段一共有 16 个字节,保存目前系统运行的一些信息)

二. 队列的实现及其管理

1. 通路的概念及其实现

限制对不可重入进程的访问,通路(gateway)是一种常用的程序结构。

不可重入过程最明显的例子是 DOS 的不可重入,最初在设计 DOS 的磁盘文件系统时,编程人员用了较多的全局变量,以标识目前文件操作的位置和状态;可是,在多任务环境下,一个低优先级任务调用 int 21h 用于写磁盘文件的操作的时候被中断,另一个优先级高的任务被激活,它也需要调用 int 21h 用于磁盘文件的操作,这样,新的状态值便代替了旧的状态值,当高优先级任务运行结束之后,低优先级任务取得控制权,这时由于相关的变量已经无法恢复,造成低优先级任务无法继续正常运转.

一个很容易想到的方法是让一个任务彻底完成使用该不可重入进程的调用之后,才允许另一个任务调用这个不可重入的进程.

gateway 能实现对不可重入进程的保护作用，在某一时刻它只允许一个任务使用这个进程。

gateway 实现对不可重入进程受控访问的简单模拟是：到大使馆签证，你必须清早去排队领取一个顺序号码,当叫到你的号时,你才能进去，否则，你只能耐心地等待。

一个有序的 gateway 对各调用任务提供了类似的裁决方法。gateway 为调用者分配一个号，若这个号与当前正在服务的号相符合，则允许调用者进入不可重入的过程。

在 gateway 初始化时，局部变量“当前标签号”和当前正在服务的号都应置为 0，这里列举了有序 gateway 的典型算法:

(1)调用 SYSDTI 禁止任务中断。

(2)将“我的标签号”置成等于“当前标签号”。(即申请一个属于自己的号码)

(3)“当前标签号”加 1。(以便下一次申请到下一个号码)

(4)调用 SYSETI 开中断。

(5)若“我的标签号”不等于“当前正在服务的号”，则调用 SYSSLP 睡眠一会,再进入(5);

(6)调用这个不可重入的过程。

(7)“当前正在服务的号”加 1。

(8)返回到调用者。

下例是用汇编语言编写的对 gateway 算法的实现。

```
;-----------------------------------------
  gateway  proc     far
           push     ax
           push     ds
           mov      ax,dseg
           mov      ds,ax
  g_10:
           call     SYSDTI                  ; prevent task switch
           mov      ax,current_ticket       ; apply my ticket
           inc      current_ticket
           call     SYSETI                  ; restore
  g_20 :
           cmp      ax,current_service      ; is my number up?
           jc       g_30                    ; if so jump...
           push     ax                      ; save my ticket
           mov      ax,1000                 ;
           call     SYSSLP                  ; sleep 1 second
           pop      ax                      ; restore my ticket
           jmp      g_20                    ; and try again...
  g_30 :
           pop      ds                     ; restore caller's DS
           pop      ax                      ; restore caller's AX
           call     non_reentrant           ; invoke the procedure
           push     ax                      ; save resulting AX
           push     ds                      ; save resulting DS

           mov      ax,dseg                 ; reclaim local DS
           mov      ds,ax                   ;
           inc      current_service         ; who's next?
           pop      ds
           pop      ax
           ret
```

这里用到 SYSSLP 调用，它的作用是让本任务挂起给定时间长度(释放对 CPU 的控制权),然后又继续往下执行;因为它涉及到定时管理和系统调度，将在后面讲述。

2. SRTX 中队列的表示及存取算法

从 SRTX 系统数据区的 230H 开始的部分是系统队列区,可以看出,系统可以有

MAX_TASK_NO个队列(可以自己定义),每个队列有108H字节,其中,后8个字节(100h-107h)是队列的控制信息,前 100h 个字节存放消息指针(offset+segment=4bytes,即双字指针,偏移量在前 2 字节,数据段值在后 2 字节),因此每个队列最多可以同时存放 100h/4=64 条指针消息.

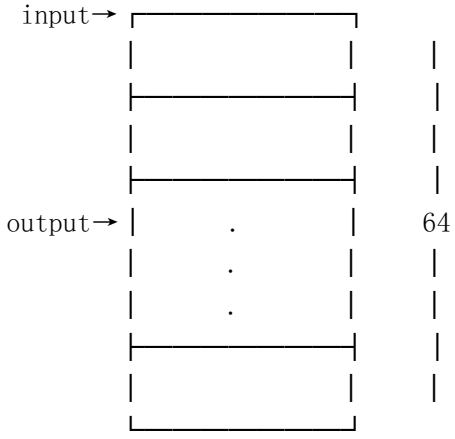
后 8 字节的详细安排如下表 3:

表 3　队列控制字段的位置和作用

偏移(字节)	作　　用
100h	本队列的最大记录数(由队列创建时指定)
101h	本队列中现有记录数(现存消息数目)
102h	访问队列的当前标签 id(见通路的描述)
103h	提供服务的 server_id(见通路的描述)
104h-105h	队列的输入指针(输入消息的存放位置:put)
106h-107h	队列的输出指针(输出消息的存放位置:out)

应该注意的是，第 100h 是一个字节，可以记录最大 255 条记录，因为本系统最大才允许有 64 条记录，因此 100h 的高两位(bit 7-6)其实没有用上，在 SRTX 中，设置 100h 的最高位为 1,表明本队列没有被使用.在初始化时,设置 100h 的内容为 80H,就表明本队列尚未被使用(程序详见" 任务状态及其表示"的[系统数据区的初始化]).

SRTX 采用循环队列的算法实现对队列的管理,循环队列实现方法如下:



其中:

DWORD buffq[64];　　//64 为队列的深度,DWORD 表示为 2 个字,第一字为 offset,第二字为 seg

ulong　input:　为输入指针,消息存放在输入指针的位置

ulong　output:　为输出指针;消息从输出指针位置取出

队列空的条件: input = output

队列满的条件: input = [output+1] mod 64;

实现队列管理的汇编代码见如下的介绍.

3. 队列的创建及其实现

在 SRTX 中，syqcre 用于创建队列，其使用方法是：

输入：寄存器 CX 包含欲创建队列的最大深度(注意必须小于最大队列深度 64).

输出：若创建成功，CARRY 清 0；此时，AX 包含了队列的 ID 号.若 CARRY 置位，AX 寄存器将包含下列错误码：

0451h……无效队列长度。

044eh……队列建立错误

以下 syqcre 是的典型使用方法：

```
;----------------------------------------
      mov     cx,40
      call    syqcre
      jc      creat_error
      mov     queue_id,AX
```

以下是 syqcre 的实现代码：

```
;----------------------------------------
;syqcre create a queue
;----------------------------------------
syqcre_sub:
        push    bx
        push    cx
        push    dx
        push    si
        push    ds                      ;reserve used regs

        mov     bx,dseg
        mov     ds,bx
        or      cx,cx                   ; is queue's size zero?
        jnz     syqcre_sub_check1       ; no jump
        mov     ax,0451h                ; queue depth=0 error
        jmp     syqcre_sub_error

syqcre_sub_check1:
        cmp     cx,40h                  ; larger than 64 ?
        jbe     syqcre_sub1             ; no jump
        mov     ax,0451h                ; error
```

```
        jmp     syqcre_sub_error

syqcre_sub1:
        mov     dl,cl                   ; depth in cl
        xor     ax,ax                   ; first queue ID=0->AX
        mov     bx,0230h
        mov     cx,max_task_no

syqcre_sub2:
        test    byte ptr[bx+0100h],80h  ; is this queue AVAILABLE?
        jnz     fill_queue_message      ; yes, jmp
        add     bx,0108h                ; find next queue
        inc     ax                      ; queue id increased
        loop    syqcre_sub2
        mov     ax,044eh                ; no empty queue for use
        jmp     syqcre_sub_error

fill_queue_message:
        mov     si,bx                   ; bx->queue buffer head
        mov     [si+0100h],dl           ; record size for this queue
        mov     byte ptr[si+0101h],0    ; initial message count
        mov     byte ptr[si+0102h],0    ; initial current label number
        mov     byte ptr[si+0103h],0    ; initial server number
        mov     ds:[si+0104h],bx        ; initial input pointer
                                        ; point to first element
        mov     ds:[si+0106h],bx        ; initial output pointer
                                        ; point to first element,too
        clc                             ;
        jmp     syqcre_sub_exit

syqcre_sub_error:
        stc

syqcre_sub_exit:
        pop     ds
        pop     si
        pop     dx
        pop     cx
        pop     bx
        ret
```

由上可见, queue_ID 与 Task_ID 的定义方法差不多,表示队列创建的顺序号(创建的第一个队列,queue_ID=0,创建的第二个队列, queue_ID=1……), 队列创建后，初始队列消息数为 0(队列中没有消息),input 和 output 指针都指向队列缓冲区的头部,

初始当前标签号和服务号均为 0.

4. 插入一条消息到指定队列

在 SRTX 中，syqpst 用于发送消息到队列，其使用方法是：

输入：寄存器 BX 包含队列的标识符,DX:AX 包含记录在队列里的双字信息.

输出：若成功，CARRY 清 0；若 CARRY 置位，AX 寄存器将包含下列错误码：

044dh……队列标识符错误。

044fh……队列满错误

以下 syqpst 是的典型使用方法：

```
;--------------------------------------
mov     ax,offset block
mov     dx,seg    block
mov     bx,queue_ID
call    SYQPST
jc      error
```

以下是 syqcre 的实现代码,注意其中对临界区和队列的操作方法：

```
;--------------------------------
;syqpst: post a message to a queue
;--------------------------------
syqpst_sub:
        push    bx
        push    dx
        push    si
        push    di
        push    ds

        push    ax
        push    dx
        mov     ax,bx
        cmp     ax,max_task_no
        jb      syqpst_sub1
        mov     ax,044dh                ;queue_id is too large
        jmp     syqpst_sub_error

syqpst_sub1:
        mov     bx,dseg
        mov     ds,bx
        mov     bx,0108h
        mul     bx
        mov     bx,ax
```

```
        add     bx,0230h                ; bx->this queue addr

syqpst_sub2:
        test    byte ptr[bx+0100h],80h  ; is queue empty?
        jz      syqpst_sub3             ; no jmp
        mov     ax,044dh                ; queue is still not created
        jmp     syqpst_sub_error

syqpst_sub3:
        call    sysdti_sub              ; disable task switch
        mov     al,[bx+0101h]           ; get current message count
        cmp     al,[bx+0100h]           ; greater than max length?
        jb      syqpst_sub4             ; no,jmp
        call    syseti_sub              ; enable task switch
        mov     ax,044fh                ; queue is full
        jmp     syqpst_sub_error        ; exit

syqpst_sub4:
        mov     si,[bx+0104h]           ; take input pointer
        pop     dx                      ; dx=seg
        pop     ax                      ; ax=offset
        mov     [si],ax                 ; insert a double word
        mov     [si+02],dx              ; message in current location
        mov     di,bx
        add     di,0100h                ; di->queue bottom
        add     si,04                   ; next input pointer
        cmp     si,di                   ; input pointer arrive at bottom?
        jb      syqpst_sub5             ; No, jump
        mov     si,bx                   ; input pointer -> queue head
;1996.1.29 by yqh
syqpst_sub5:
        mov     [bx+0104h],si           ; reserve input pointer
        inc     ds:byte ptr [bx+0101h]  ; message count++
        call    syseti_sub              ; enable task switch again
        clc
        jmp     syqpst_sub_exit

syqpst_sub_error:
        pop     bx
        pop     bx
        stc

syqpst_sub_exit:
        pop     ds
```

```
        pop     di
        pop     si
        pop     dx
        pop     bx
        ret
```

注意其中对队列输入指针的处理,每加入一条消息,输入指针 input 加 4,若输入指针加 4 后已经到达队列的底部,这时需要将输入指针调整到队列的头部,实现循环队列的操作.

5. 从队列中取出一条消息(1)

在 SRTX 中，syqacc 用于从队列中取出一条消息，其使用方法是：

输入：寄存器 BX 包含队列的标识符.

输出：若成功，CARRY 清 0；DX:AX 中包含双字信息,若 CARRY 置位，AX 寄存器将包含下列错误码：

044dh……队列标识符 ID 错误。

0450h……队列空错误.

以下 syqacc 是的典型使用方法：

```
;--------------------------------------
    mov    bx,  queue_ID
    call   SYQACC
    jc     error
    mov    ds,  dx
    mov    si,  ax
```

以下是 syqacc 的实现代码, 注意其中对临界区和队列的操作方法：

```
;--------------------------------------
;syqacc accept a message from a queue
;--------------------------------------
syqacc_sub:
        push    bx
        push    cx
        push    si
        push    di
        push    ds

        mov     ax,bx
        cmp     ax,max_task_no
        jb      syqacc_sub1
        mov     ax,044dh                ;queue_ID is to large
```

```
        jmp     syqacc_sub_error

syqacc_sub1:
        mov     bx,dseg
        mov     ds,bx
        mov     bx,0108h
        mul     bx
        mov     bx,ax
        add     bx,0230h                ; bx-> this queue buffer
        test    byte ptr [bx+0100h],80h ; is this queue created ?
        jz      syqacc_sub2             ; yes, jump
        mov     ax,044dh                ; queue is still not created
        jmp     syqacc_sub_error

syqacc_sub2:
        call    sysdti_sub
        mov     al,[bx+0101h]           ; get msg count
        or      al,al                   ; is there message in queue
        jnz     syqacc_sub3             ; yes jump
        call    syseti_sub              ; no msg in queue
        mov     ax,0450h                ;
        jmp     syqacc_sub_exit

syqacc_sub3:
        mov     si,[bx+0106h]           ; remove a message from queue
        mov     ax,[si]                 ; get offset first
        mov     dx,[si+02]              ; get seg
        add     si,04                   ; next output pointer
        mov     di,bx
        add     di,0100h                ; queue bottom
        cmp     si,di                   ; if next point to bottom?
        jb      syqacc_sub4             ; no, jump
        mov     si,bx                   ; next point to queue head

syqacc_sub4:
        mov     [bx+0106h],si           ; set next output pointer
        dec     byte ptr [bx+0101h]     ; dec message count
        call    syseti_sub
        clc
        jmp     syqacc_sub_exit

syqacc_sub_error:
        stc
```

```
syqacc_sub_exit:
        pop     ds
        pop     di
        pop     si
        pop     cx
        pop     bx
        ret
```

注意其中对队列输出指针的处理,每取出一条消息,输出指针 output 加 4,若输出指针加 4 后已经到达队列的底部,这时需要将输出指针调整到队列的头部,实现循环队列的操作.

注意,SYQACC 实现的是从队列中直接取出一条记录,若队列中没有消息,则返回错误代码 0450H,以下要介绍的 SYQPND 也是从队列中取出一条记录,但若队列中没有消息时,本任务需要挂在此队列上等待,直到超时为止.(由于 SYQPND 涉及到任务调度和定时管理的内容,因此留到后面的章节来介绍).

6. 队列状态查询

在 SRTX 中，syqinq 用于查询指定队列的状态，其使用方法是：

输入：寄存器 BX 包含队列的标识符.

输出：若 CARRY 为 0；CX 包含本队列中的元素数目(可能为 0)，DX:AX 中包含双字信息(当 CX=0 时,DX:AX 无效);若 CARRY 置位，查询失败,CX=0,AX 寄存器将包含下列错误码：

044dh……队列标识符 ID 错误。

以下是 syqinq 的典型使用方法：

```
;-------------------------------------

    mov    bx,  queue_ID
    call   SYQINQ
    jc     error
    or     cx,  cx
    jz     no_mail
    mov    ds,  dx
    mov    si,  ax
```

以下是 syqinq 的实现代码,注意其中对临界区的操作：

```
;--------------------------------
;syqinq perform a queue inquiry
;--------------------------------
syqinq_sub:
        push    bx
```

```
        push    si
        push    di
        push    ds

        xor     cx,cx
        mov     ax,bx
        cmp     ax,max_task_no
        jb      syqinq_sub1
        mov     ax,044dh                ;queue_ID is too large
        jmp     syqinq_sub_error

syqinq_sub1:
        mov     bx,dseg
        mov     ds,bx
        mov     bx,0108h
        mul     bx
        mov     bx,ax
        add     bx,0230h                ; bx-> queue head
        test    byte ptr[bx+0100h],80h  ; is queue created ?
        jz      syqinq_sub2             ; yes, jump
        mov     ax,044dh                ; queue is not still created
        jmp     syqinq_sub_error

syqinq_sub2:
        call    sysdti_sub
        mov     si,[bx+0106h]           ; find out addr of message
        mov     ax,[si]                 ; dx:ax contain a copy of
        mov     dx,[si+02]              ; first_out queue data
        mov     cl,[bx+0101h]           ; take message count
        call    syseti_sub
        clc
        jmp     syqinq_sub_exit

syqinq_sub_error:
        stc

syqinq_sub_exit:
        pop     ds
        pop     di
        pop     si
        pop     bx
        ret
```

注意 SYQINQ 完成的操作只是将双字指针的拷贝取出,并不象 SYQACC 那样将输出指针移位,在 SYQINQ 之后,再用 SYQACC,得到的 DX:AX 与 SYQINQ 完全一样.

三. 定时管理及其实现

多任务程序的超时机制和状态的切换都与定时管理有关，本节介绍以下 SRTX 的定时管理。

1．定时器及定时中断

多任务程序的运行一定需要有定时机制的硬件支持，通过对硬件的编程，可以使系统实现既定时间长度产生中断，在中断服务程序中，完成对相应需要定时的进程进行控制。

SRTX 是针对 PC 机而设计的，DOS 系统每隔 55ms 产生一次 INT8 的中断，在 INT8 的中断服务程序中，调用了 INT 1C(INT 1C 的函数体是空),因此可以编一个自己的函数替换 INT1C,就可以让系统定时调用我们自己的函数，在此函数中,可以让它实现定时管理功能,在 SRTX 中,SYSRTI 就是这样一个函数.

2．定时器的实现

在 SRTX 中,系统定时调用 sysrti,实现定时管理功能.以下是 SYSRTI 的实现代码:

```
;--------------------------------
;colck interrupt handler
;--------------------------------
sysrti  proc    far
        push    ds
        push    ax
        mov     ax,dseg
        mov     ds,ax
        add     ds:word ptr[04],55      ; inc system clock

        test    ds:byte ptr[07],0ffh    ; task can switch ?
        jnz     sysrti_sub2             ; not,exit
        test    ds:byte ptr[0bh],0ffh   ; task has been suspended?
        jz      sysrti_sub2             ; yes,exit
        jmp     sysrti_sub3             ;

sysrti_sub2:
        pop     ax                      ; direct exit
        pop     ds
        sti
        iret

sysrti_sub3:
```

```
        push    bx
        mov     bx,ds:[0eh]             ; take out TCB for running task
        or      byte ptr[bx+01],10h     ; set interrupted flag
        pop     bx
        pop     ax                      ; reserve all state

        push    es                      ; for recover running
        push    di                      ; ( see module of
        push    si                      ; run_interrupt_task )
        push    bp                      ;
        push    bx
        push    dx
        push    cx
        push    ax                      ; reserve environment
        mov     bx,ds:[0eh]             ; take out TCB pointer
        mov     [bx+0ah],sp             ; save current task status
        mov     [bx+0ch],ss
        mov     ds:byte ptr[0bh],0      ; set suspended  flag
        jmp     reschedul               ; enter next schedul
sysrti  endp
```

在本程序中,用到几个系统变量:

[04-05]:字变量,用于表示从本程序运行以来经过的 ms 数,因为每隔 55ms 调用一次,因此[04-05]每次加 55.

[07]:当前运行的任务是否可以被切换.因为硬件中断是随时可以发生的,在 SYSRTI 中需要根据任务的状态随时将最高优先级的任务取出运行,若当前运行的任务不允许被切换,设置[07]为 0xff.(见上面相应章节的说明)

[0bh]:当前任务被挂起的标志,若任务被挂起,设置[0bh]=0,不允许进行任务的切换。(这标志是否有必要?)

[0eh-0fh]:当前运行任务的 TCB 指针.

可以看出,SYSRTI 的最后,取出当前任务的TCB,将断点及现场都保存在任务的堆栈中,把堆栈指针保存在本任务 TCB 的[0a-0d]字节中(这里可以看出 TCB 中的[10-11,12-13]字节的作用),然后进入调度程序(reschedul).

3．任务睡眠(SYSSLP)及其实现

在 SRTX 中，sysslp 用于让任务挂起定长时间，其使用方法是：

输入：寄存器 AX 包含挂起时间长度(ms).

输出：无.

以下 sysslp 的典型使用方法：

```
;---------------------------------
     MOV   DX,PARALLEL_PORT
P1:
     IN    AL,  DX
     TEST  AL,  READY_BIT
     JNZ   DO_OUTPUT
     MOV   AX,  55
     CALL  SYSSLP
     JMP   P1
```

以下是 SYSSLP 的实现代码：

```
;---------------------------------
;sysslp sleep for specfied duration
;---------------------------------
sysslp_sub:
        cli                             ; forbit clock interrupt
        call    current_time_agument    ; time calculate
        call    suspend_itself          ; set task suspended
        ret                             ; enter next schedul
;-------------------------------
current_time_agument:
        push    ax
        push    bx
        push    cx
        push    ds
        mov     bx,dseg
        mov     ds,bx
        mov     bx,ds:[0eh]             ; take out TCB
        mov     cx,ds:[04]              ; get current clock
        add     ax,cx                   ; current time add agument
        mov     [bx+0eh],ax             ; task will be awakened
        mov     al,[bx+01]              ; at 'ax' moment
        and     al,0fdh                 ; set not awakened status
        or      al,04                   ; set sleep status
        mov     [bx+01],al              ; save
        pop     ds
        pop     cx
        pop     bx
        pop     ax
        ret
;---------------------------------
;suspend current task
;---------------------------------
```

```
suspend_itself  proc    far
        pushf                           ; called only by sysslp_sub
        push    ds
        push    es                      ; reserve current status
        push    di                      ; for awakening to run
        push    si
        push    bp
        push    bx
        push    dx
        push    cx
        push    ax                      ; reserve enviroment
        mov     ax,dseg
        mov     ds,ax
        mov     ds:byte ptr[0bh],0      ; set suspended  flag
        mov     bx,ds:[0eh]             ; take TCB addr of current task
        mov     [bx+0ah],sp             ; reserve stack address
        mov     [bx+0ch],ss             ; (all status is in stack)
        jmp     reschedul               ; enter next schedul
suspend_itself  endp
```

这里终于可以看出在 TCB 中的 14-15 字节(slice)的作用了,SYSSLP 的 AX 值与当前时钟值相加,放入 slice 中.从今后的相关代码中可以看出,slice 时刻一到,任务就被唤醒.

suspend_itself 函数将当前任务的现场保存在其 TCB 中(10-11,12-13 字节:SP 和 SS),然后进入调度程序.

4. 从队列中取消息 SYQPND 的实现.

以前我们讲了从队列中取消息 SYQACC 的实现,以下我们讲 SYQOND 的实现.

输入：寄存器 BX 包含队列的标识符,AX 包含了若没有消息时最多等待的 ms 数.

输出：若 CARRY 为 0； DX:AX 中包含从该队列中取得的双字信息;若 CARRY 置位，AX 寄存器将包含下列错误码：

044dh……队列标识符 ID 错误。

0450h……超时错误。

以下是 syqpnd 的典型使用方法：

```
;---------------------------------

        mov     bx,   queue_ID
check:
        mov     ax,   1000
        call    SYQPND
        jnc     OKEY
```

```
    jmp    check
```

以下是 syqpnd 的实现代码:

```
;--------------------------------------
;syqpnd: receive a message from queue
;--------------------------------------
syqpnd_sub:
        push    bx
        push    cx
        push    si
        push    di
        push    ds
        push    ax
        mov     ax,bx
        cmp     ax,max_task_no
        jb      syqpnd_sub01            ;queue_ID is too large
        mov     ax,044dh
        jmp     syqpnd_sub_error

syqpnd_sub01:
        mov     bx,dseg
        mov     ds,bx
        mov     bx,0108h

syqpnd_sub1:
        mul     bx
        mov     bx,ax
        add     bx,0230h                        ; bx->this queue head
        test    ds:byte ptr[bx+0100h],80h       ; is queue created?
        jz      syqpnd_sub2                     ; yes, jump
        mov     ax,044dh                        ; queue is not created
        jmp     syqpnd_sub_error

syqpnd_sub2:
        call    sysdti_sub
        mov     ch,[bx+0102h]           ; get current pass id
        inc     ds: byte ptr[bx+0102h]  ; next pass id
        call    syseti_sub
        mov     cl,01

syqpnd_sub22:                           ; my pass id is equal to
        cmp     ch,[bx+0103h]           ; current server id ?
        je      syqpnd_sub3             ; yes jump
        or      cl,cl                   ; time out ?
        jz      syqpnd_sub6             ; yes, exit
        jmp     syqpnd_sub4             ; no, delay ax ms

syqpnd_sub3:
        test    byte ptr [bx+0101h],0ffh; is there message?
        jnz     syqpnd_sub7             ; yes,jump
```

```
        or      cl,cl                   ; time out ?
        jz      syqpnd_sub5             ; yes,exit

syqpnd_sub4:
        pop     ax
        push    ax
        call    sysslp_sub              ; delay AX ms
        dec     cl                      ; set time out flag
        jmp     syqpnd_sub22            ; continue fetch message

syqpnd_sub5:
        inc     byte ptr [bx+0103h]     ; set next server id

syqpnd_sub6:
        mov     ax,0450h                ; time out error
        jmp     syqpnd_sub_error

syqpnd_sub7:
        pop     ax
        call    sysdti_sub
        mov     si,[bx+0106h]           ; take out a message
        mov     ax,[si]
        mov     dx,[si+02]
        add     si,04                   ; form next output pointer
        mov     di,bx
        add     di,0100h                ; queue bottom
        cmp     si,di                   ; is point to queue bottom
        jb      syqpnd_sub8             ; no, jump
        mov     si,bx                   ; yes,next output pointer
                                        ; is queue head
syqpnd_sub8:
        mov     [bx+0106h],si           ; save output pointer
        dec     byte ptr [bx+0101h]     ; dec msg number
        inc     byte ptr [bx+0103h]     ; next server id
        call    syseti_sub              ; this is gateway alg
        clc
        jmp     syqpnd_sub_exit

syqpnd_sub_error:
        pop     bx
        stc

syqpnd_sub_exit:
        pop     ds
        pop     di
        pop     si
        pop     cx
        pop     bx
        ret
```

在 syqpnd 中,若队列中没有消息,调用了 sysslp,这期间,本任务放弃了对 CPU 的

控制权,其它任务被激活运行,它们可能往本队列中 syqpst 消息,这样,当本任务超时恢复运行时,就可以取出该消息了.

四. 任务调度及其实现

任务调度是实时多任务程序的核心,它的主要功能是从当前众多可运行的任务中提取最高优先级的可运行任务,找到该任务的 TCB 地址,从该 TCB 中取出上次被中断运行的现场,从而将该任务投入运行.

首先介绍几个函数:

take_ptr_pair:功能是提取指定位置的双字指针.

输入:AX=欲提取的双字指针的偏移位置.

输出:AX:BX=双字指针

如:

```
        mov     ax,0ah ;should be 06h
        call    take_ptr_pair           ; take out current ss:sp
        mov     ss,ax                   ; of this awakened task
        mov     sp,bx
```

以上语句的功能是提取一个任务上次被中断是的 SS:SP,这样可以恢复该任务上次运行的断点. 因为:TCB 的 0ah 位置存放的是上次被中断时的 SS:SP.若 AX=04H,表示提取任务初始的 CS:IP.

take_ptr_pair 的实现代码如下:

```
;------------------------------
take_ptr_pair:
        pushf
        mov     bx,ds:[0eh]
        add     bx,ax                   ; get TCB
        mov     ax,[bx+02]              ; take out a pointer pair
        mov     bx,[bx]                 ;
        popf
        ret
```

task_not_run:功能是挂起本任务,设置本任务为不可运行状态.

task_not_run 的实现代码如下:

```
;----------------------------------------
;stop a task,this task will not run forever
;----------------------------------------
task_not_run:
```

```
        mov     ax,dseg
        mov     ds,ax
        mov     ds:byte ptr[0bh],0      ; set suspend flag
        mov     bx,ds:[0eh]             ; take out TCB pointer
        mov     byte ptr[bx+01],0       ; stop task running
        jmp     reschedul               ; task is excluded from reschedul
```

set_run_flag:功能是设置任务未运行结束,未被挂起状态.

set_run_flag 的实现代码如下:

```
;------------------------------
set_run_flag:
        push   ax
        mov    al,0ffh                  ; set current task
        mov    ds:[0bh],al              ; not be suspended
        mov    ds:[0ch],al              ; current task is
        pop    ax                       ; not ended
        ret
```

run_ready_task:功能是将一个处于 READY 状态的任务投入运行.

一般来说,任务在创建之后,才处于 READY 状态,这时候,任务还没有运行过,因此任务必须从头运行.在任务运行中间,它可能被中断(即定时中断 SYSRTI 将此任务置为被中断的状态),也可能由于等待资源而处于挂起(睡眠)状态,但是任务只要投入运行,它就不会处于 READY 状态.

run_ready_task 的实现代码如下:

```
;-------------------------------------------
;pass cpu control to task that is ready status
;-------------------------------------------
run_ready_task  proc    far
1       mov     ax,06
2       call    take_ptr_pair           ; take out initial ss:sp
3       mov     ss,ax                   ; form task stack
4       mov     sp,bx
5       call    set_run_flag            ; set task is ready
6       push    cs                      ; cs of task_not_run
7       mov     ax,offset task_not_run  ; ip of task_not_run
8       push    ax                      ; (after running this
9       mov     ax,02                   ; task completely,
10      call    take_ptr_pair           ; run task_not_run)
11      push    ax                      ; take out initial cs:ip
```

```
12          push    bx
13          ret                                 ; start to run this task
run_ready_task  endp
```

此程序有点技巧,1-4 行,提取任务创建时的 SS:SP 作为目前的 SS:SP,6-8 行,将函数 task_not_run 的 CS:OFFSET 压入堆栈,9-12 行,将任务创建时的 CS:OFFSET 提取出来并压入堆栈,13 行的 ret 指令实现函数的返回,因为 ret 指令要作出栈操作,因此将刚压入堆栈的 CS:OFFSET 弹出,作为 ret 指令之后的 CS:IP,因此任务投入运行.本任务在彻底运行完成以后,与以上相同的机制弹出 task_not_run 的 CS:IP,于是 task_not_run 开始运行.

该编程技巧经常用于程序的反跟踪,需要仔细领会.

run_awakened_task: 功能是将一个处于睡眠超时状态的任务投入运行.

程序首先取出上次挂起时的 SS:SP,然后从该堆栈中取出上次压入的各寄存器的值,恢复现场并返回.注意在 reschedul 中采用的是 jmp run_awakened_task 指令而不是采用 CALL run_awakened_task 指令是为了与 suspend_itself(在 SYSSLP_SUB 中调用)中 jmp reschedul 相呼应, run_awakened_task 的末尾用 ret 指令就返回到调用 SYSSLP 的下一条指令.

run_awakened_task 的实现代码如下:

```
;-------------------------------------------------
;pass cpu control to task that is awakened status
;-------------------------------------------------
run_awakened_task       proc    far
        mov     ax,0ah
        call    take_ptr_pair           ; take out current ss:sp
        mov     ss,ax                   ; of this awakened task
        mov     sp,bx
        call    set_run_flag            ; set task is not suspended
        pop     ax                      ; restore all status
        pop     cx                      ; and flags,
        pop     dx                      ; continue running
        pop     bx                      ; task from awakend
        pop     bp                      ; address,usually,
        pop     si                      ; this address is after
        pop     di                      ; sysslp call.
        pop     es                      ; for example:
        pop     ds                      ;       mov     ax,1000
        popf                            ;       call    sysslp
```

```
        sti                             ;  ->   ;here is awakened addr
        ret                             ;       mov     si,bx
run_awakened_task       endp            ; (pushed by suspend_itself)
```

run_interrupt_task: 功能是将一个处于被中断状态的任务投入运行.

程序首先取出上次被中断时的 SS:SP,然后从该堆栈中取出上次压入的各寄存器的值,恢复现场并返回.注意在 reschedul 中采用的是 jmp run_interrupt_task 指令而不是采用 CALL run_interrupt_task 指令是为了与 sysrti(在 SYSSLP_SUB 中调用)中 jmp reschedul 相呼应, run_interrupt_task 的末尾用 iret 指令表明中断服务程序的返回.

run_interrupt_task 的实现代码如下:

```
;-------------------------------------------------
;pass cpu control to task that is interrupted
;-------------------------------------------------
run_interrupt_task      proc    far
          mov     ax,0ah
          call    take_ptr_pair          ; take out current ss:sp of
          mov     ss,ax                  ; this interrupted task
          mov     sp,bx                  ; set new ss:sp of this task
          call    set_run_flag           ; set task is not suspended
          pop     ax                     ; restore break point
          pop     cx                     ; of interrupted task
          pop     dx
          pop     bx
          pop     bp                     ; interrupt break point is
          pop     si                     ; pushed by sysrti -real
          pop     di                     ; time manage program
          pop     es
          pop     ds                     ; restore to run from
          iret                           ; interrupted point
run_interrupt_task      endp
```

读懂以上几个函数以后, reschedul 程序就不难理解了.

以下是任务调度的实现代码,从 reschedul 开始:

```
;-------------------------------------------------------------
schedul:
        mov     bx,ds:[0eh]             ; form currnt tcb address
        add     bx,10h                  ; point to next TCB
        cmp     bx,0220h                ; has finished ?
```

```
        jb      reschedul1              ; no, check status
        test    ds:byte ptr[0ch],0ffh   ; task execute end during
        mov     ds:byte ptr[0ch],0      ; this tick?
        jnz     reschedul               ; no, jump
        sti                             ; important,by yqh
        hlt                             ; wait for next interrupt

reschedul:                              ; entry for reschedul procedure
        cli                             ; important,by yqh
        mov     bx,20h                  ; bx->first TCB

reschedul1:
        mov     ds:[0eh],bx             ; reserve tcb pointer
        mov     al,[bx+01]              ; get status
        test    al,20h                  ; tcb is empty?
        jnz     schedul                 ; yes, jump
        and     al,81h                  ; is inhibited  to execute?
        xor     al,01                   ; is not runnable?
        jnz     schedul                 ; yes, jump
        mov     al,[bx+01]              ; get status
        and     al,10h                  ; task is interrupted ?
        jnz     int_process             ; yes,jump
        mov     al,[bx+01]              ; get status
        and     al,04                   ; task is seleeping?
        jnz     sleep_process           ; yes, jump
        mov     byte ptr [bx+01],01     ; set runnable status
        sti                             ; important, by yqh
        jmp     run_ready_task

int_process:
        and     byte ptr [bx+01],0efh   ; cancel interrupted status
        sti                             ; important,by yqh
        jmp     run_interrupt_task      ; entery task interrupted

sleep_process:
        mov     ax,ds:[04]              ; take out current clock
        cmp     ax,[bx+0eh]             ; cmp with awakening moment
        js      schedul                 ; not time out, jump
        mov     byte ptr [bx+01],03     ; set awakened status
        sti                             ; important,by yqh
        jmp     run_awakened_task       ; entry awakened task
```

从如上代码可以看出，任务调度总是从 reschedul 开始，而 reschedul 又总是从

第一个 TCB 开始考察各任务的状态,选出第一个可以运行的任务投入运行,由此可见,最先建立的任务的优先级最高,TCB 处于较前面的位置,由此实现基于优先级的调度.

任务调度的时机:

任务什么时候进行调度?从以上程序可以看出,任务调度发生在两个地方,一个是在函数 suspend_itself 中,另一个是 sysrti 的定时器管理程序.

suspend_itself 由 sysslp 来调用,sysslp 置任务于挂起状态.通常,任务在访问资源时,若条件不满足,需要等待资源的到来,这时通过调用 sysslp,置本任务于睡眠状态,让调度程序激活其它任务运行,当调用 sysslp 的任务超时之后,本任务恢复运行;经过这段时间,本任务需要的资源可能已经到来,因此,可以取出资源,任务继续执行,最典型的情况请见 syqpnd 程序.

sysrti 无疑是最重要的调度程序源,定时器由硬件管理,每个 tick(通常 10ms,DOS 系统是 55ms)产生一个定时中断,在中断服务程序中把最高优先级的可运行的任务提取出来,投入运行,正是因为有了这个定时器中断,才使得实时调度成为可能,同时也使得系统定时管理变得简单和容易.

五. 小结

本文提供的 SRTX 的实现只是用以说明实时多任务系统的运行原理和内部机制,实际上,一个商用的实时多任务系统比 SRTX 复杂得多,功能也强大得多.

本文附带的 SRTX 的所有原码及注释(srtx.arj),读者可以用 masm 及 link 处理,得到一个最简单的 SRTX 的演示程序,另附原版代码及演示程序(pax.arj,注意到所有 ASM 都是 1986 年的文件,确实古老).