操作系统　第四章　进程管理

4.3 时钟中断 和 时间片轮转调度

同济大学计算机系　邓蓉

1、时钟（硬件）

- RTC（ Real Time Clock ） 实时钟，在主板上，电池驱动。关机时，维护系统时间。
- PIT（ Programmable Interval Timer ）周期性中断时钟。是连在中断控制器上的外设芯片。系统工作时的主时钟。 8253时钟芯片。
- TSC（Time Stamp Counter）。是位于CPU里面的一个64位的TSC寄存器。每个CPU时钟周期，值加1。非常高精度的时钟。

时钟的用途

- 内核初始化时，读RTC的值，写全局变量 time。time是系统时钟，是wall clock time。
- 系统正常工作时，RTC不工作。 PIT为系统提供均匀的脉冲信号。这就是**时钟中断**。是计算机系统的心跳。时钟中断间隔是tick，时钟滴答。

  系统对时钟中断计数，调整time变量的值，实施与时间有关的任务。
- TSC，用来实现高精度定时器 + 修正time误差。

2、时钟中断处理程序要做的工作

维护系统时间 time

实现时间片轮转调度，公平对待所有应用程序

定时启动内核的系统维护任务

为应用程序提供定时器服务

Unix V6++的时钟中断

- 时钟中断入口程序　　硬件现场保护、中断入口程序构造中断栈帧
- 计时 （累加所有时钟脉冲计数器）
- 调整 系统时钟变量（time）
- 修正所有用户态进程的优先权
- 衡量现运行进程有没有用完时间片（RunRun++）
- 为应用程序提供定时器服务
- 如果有未处理的信号，现运行进程处理信号
- 将盘交换区上的就绪进程搬回内存
- 时钟中断入口程序　　例行调度，判断RunRun。用完时间片的现运行进程会让出CPU

减小时钟中断的运行开销，提升系统性能

- 时钟中断入口程序　　硬件现场保护、中断入口程序构造中断栈帧
- **计时 （累加所有时钟脉冲计数器），每次时钟中断都做。**
- **整数秒，若先前是用户态，做一次。**
  - **调整 系统时钟变量（time）**
  - **修正所有用户态进程的优先权**
  - **衡量现运行进程有没有用完时间片（RunRun++）**
  - **为应用程序提供定时器服务**
  - **如果有未处理的信号，现运行进程处理信号**
  - **将盘交换区上的就绪进程搬回内存**

这是为了防止耗时的操作
耽搁被中断的内核任务

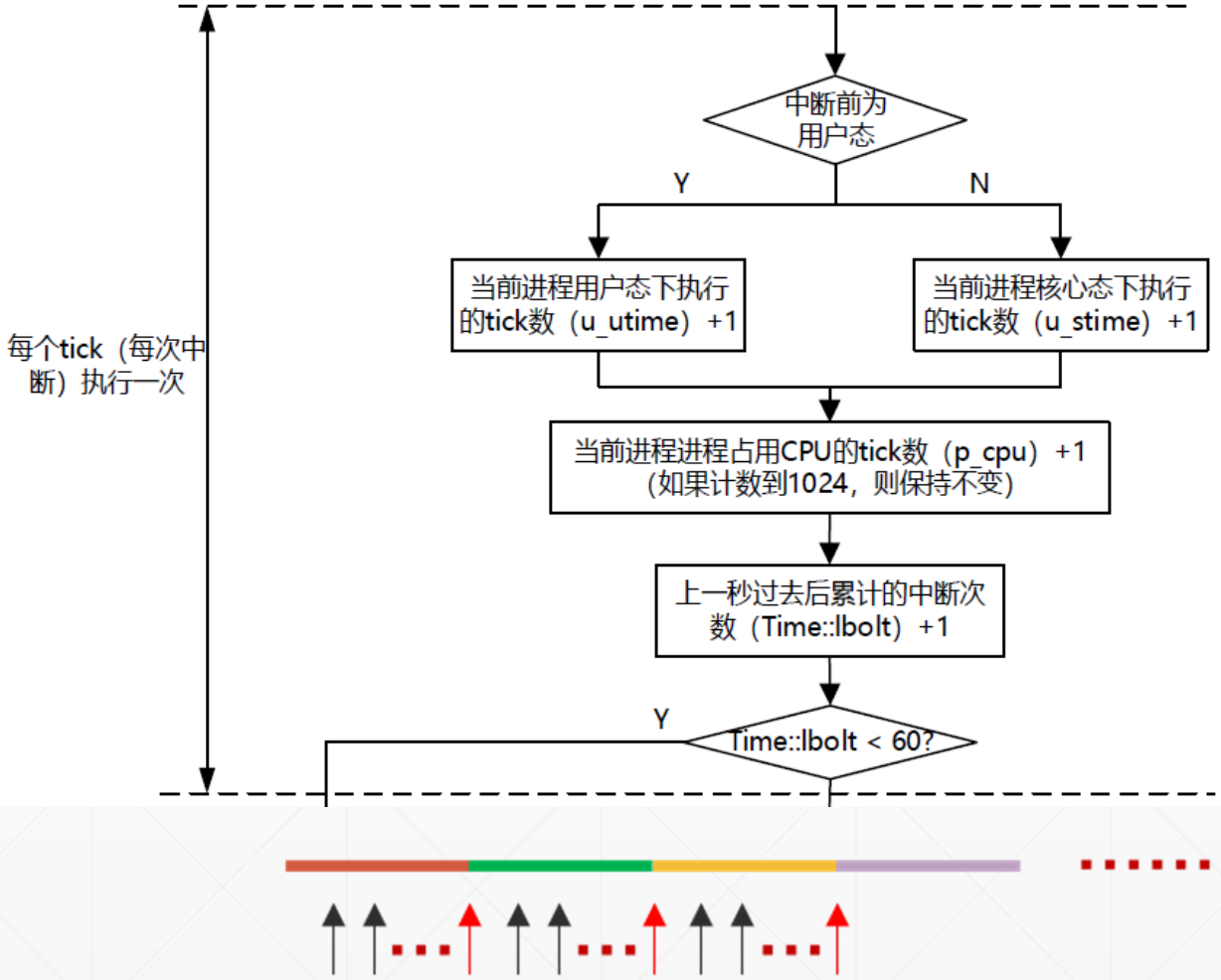
- 时钟中断入口程序　　**例行调度，判断RunRun。用完时间片的现运行进程会让出CPU**

3、UNIX V6++系统的时钟中断　　Time对象

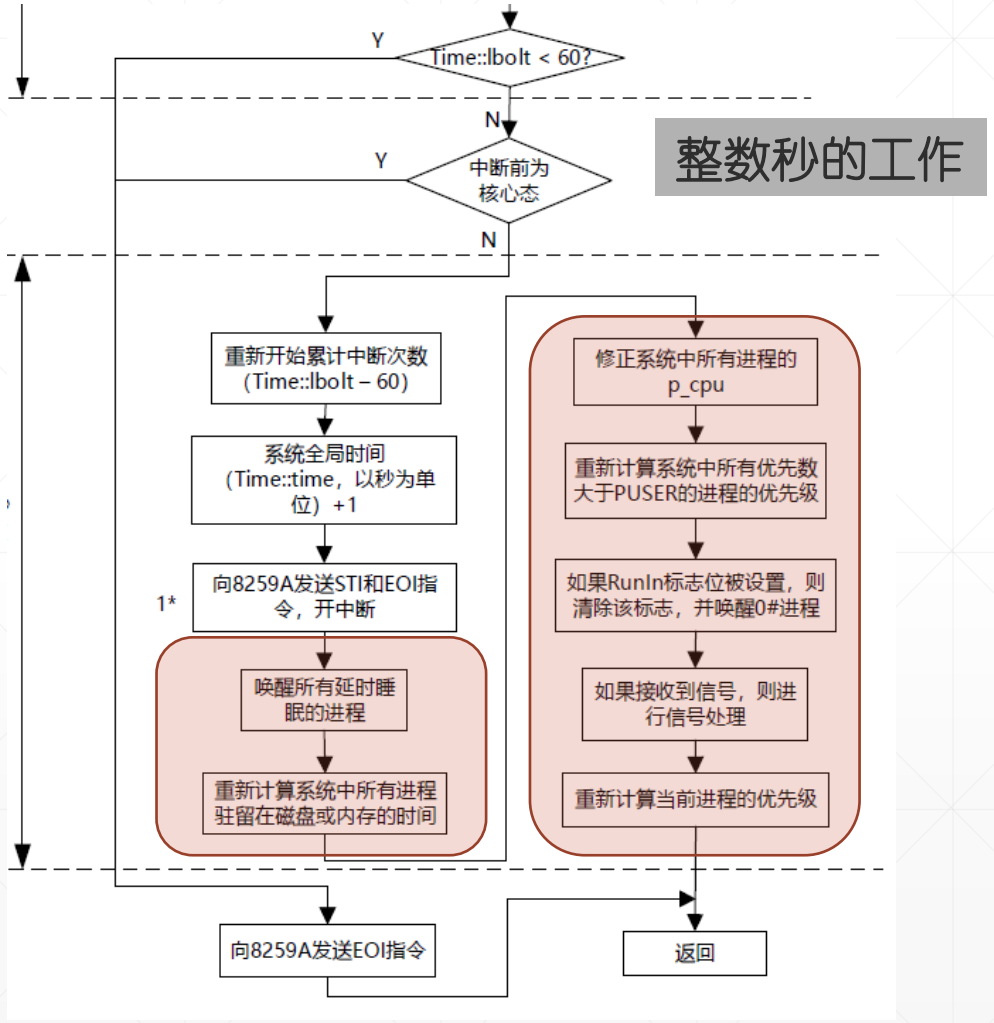
```
class Time                          // Unix系统时钟管理器。整个系统只有一个Time对象
{
    public:
    static const int SCHMAG = 10;           /* 每秒钟减少的p_cpu魔数 */
    static const int HZ = 60;               /* 每秒钟时钟中断次数 */

                                    1/HZ，一个tick（时钟滴答）。计算机系统的计时单位

    static int lbolt;                       /* 时钟滴答的计数器 */
    static unsigned int time;               /* 系统时间，自1970年1月1日至今的秒数 */
    static unsigned int tout;           /* 设置定时器的睡眠进程，所有唤醒时刻的最小值 */

    /* 时钟中断入口函数*/
    static void TimeInterruptEntrance();
    /* 时钟中断处理函数*/
    static void Clock(struct pt_regs* regs, struct pt_context* context);
};
```

3.1 时钟中断处理

每个tick进行的计数操作



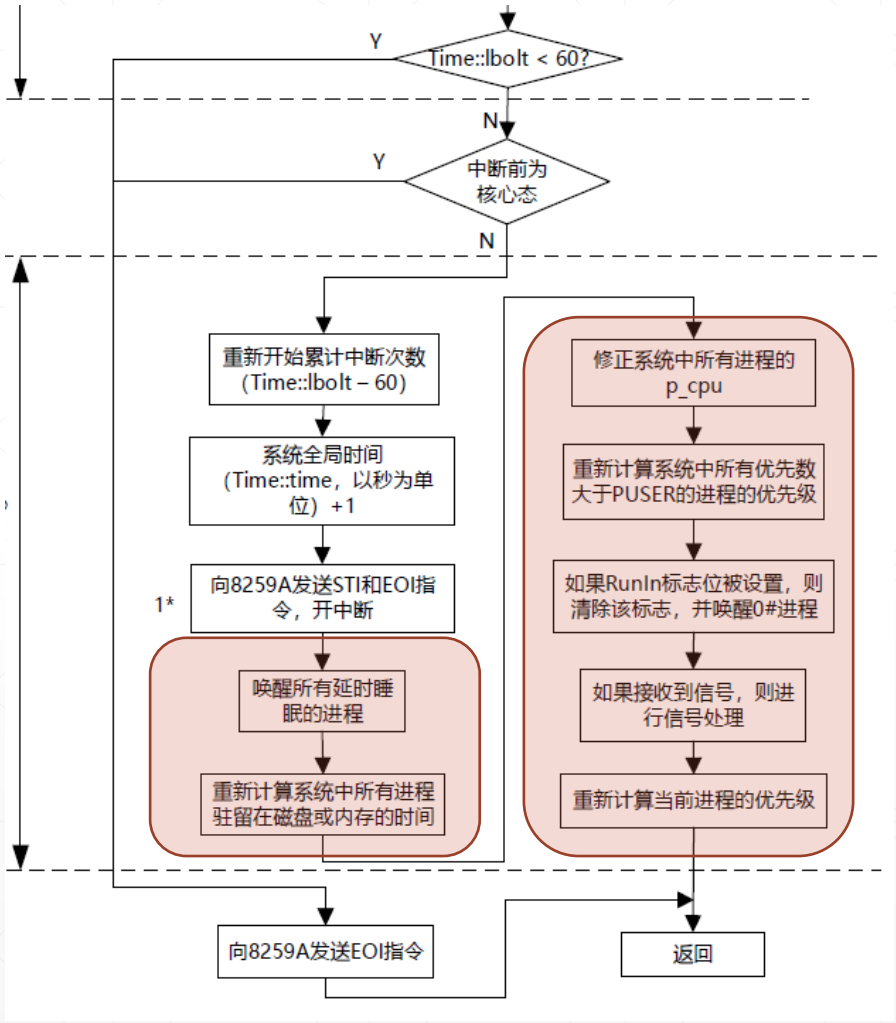
整数秒的工作



```
void Time::Clock( struct pt_regs* regs, struct pt_context* context )
```

```
if ( (context->xcs & USER_MODE) == USER_MODE )
{
        u.u_utime++;    // 当前进程用户态下执行的时间 ++
}
else
{
        u.u_stime++;    // 当前进程核心态下执行的时间 ++
}

current->p_cpu = Utility::Min(++current->p_cpu, 1024);   // 进程时钟中断次数计数器 ++
```

```
if ( ++Time::lbolt < HZ )
{
        IOPort::OutByte(Chip8259A::MASTER_IO_PORT_1, Chip8259A::EOI);
        return;
}
```



整数秒的系统维护操作（预备）

```
if( current->p_stat == Process::SRUN &&
  (context->xcs & USER_MODE) == KERNEL_MODE )
{
        发EOI命令;
        return;
}

Time::lbolt -= HZ;
Time::time++; //修改wall clock time

X86Assembly::STI();
IOPort::OutByte(Chip8259A::MASTER_IO_PORT_1, Chip8259A::EOI);
```

3.2 时钟中断处理 ——整数秒的系统维护操作

1、唤醒设置了定时器的进程
也就是，执行系统调用sleep（seconds）入睡的进程

2、所有进程，p_time++， p_cpu - 10
重新计算SRUN进程的优先数
系统调用的优先数不动。

```
if ( Time::time == Time::tout )
{
    /* 唤醒延时睡眠的进程 */
    procMgr.WakeUpAll((unsigned long)&Time::tout);
}
-------------------------------------------------------------------------------
/* 重算所有进程的p_time, p_cpu,以及优先数p_pri */
for( int i = 0; i < ProcessManager::NPROC; i++ )
{
    Process* pProcess = &procMgr.process[i];
    if ( pProcess->p_stat != Process::SNULL )
    {
        pProcess->p_time = Utility::Min(++pProcess->p_time, 127);

        if ( pProcess->p_cpu > SCHMAG )
        {
            pProcess->p_cpu -= SCHMAG;
        }
        else
        {
            pProcess->p_cpu = 0;
        }
        if ( pProcess->p_pri > ProcessManager::PUSER )
        {
            pProcess->SetPri();

        }
    }
}
```

// 进程在内存或盘交换区的驻留时长

3、如果盘交换区有就绪进程，唤醒0#进程，试一下可不可以把它们搬回内存。

4、现运行进程处理信号。

5、如果时间片用完，RunRun标识置1。

```
    if ( procMgr.RunIn != 0 )
    {
        procMgr.RunIn = 0;
        procMgr.WakeUpAll((unsigned long)&procMgr.RunIn);
    }

    /* 如果中断前为用户态，则考虑进行信号处理 */
    if ( (context->xcs & USER_MODE) == USER_MODE )
    {
        if ( current->IsSig() )
        {
            current->PSig(context);
        }
        /* 计算当前进程优先数 */
        current->SetPri();
    }
}
```

3.3 时钟中断入口程序——中断返回

```
if( context->xcs & USER_MODE )      /*先前为用户态*/
{
        while(true)
        {
                X86Assembly::CLI();
                if(Kernel::Instance().GetProcessManager().RunRun > 0)
                {
                        X86Assembly::STI();
                        Kernel::Instance().GetProcessManager().Swtch();       // 开中断放弃CPU
                }
                else
                {
                        break;   /* 如果runrun == 0，则退栈回到用户态继续执行用户程序 */
                }
        }
}
```

恢复现场，返回

例题1：描述时钟中断处理程序的行为

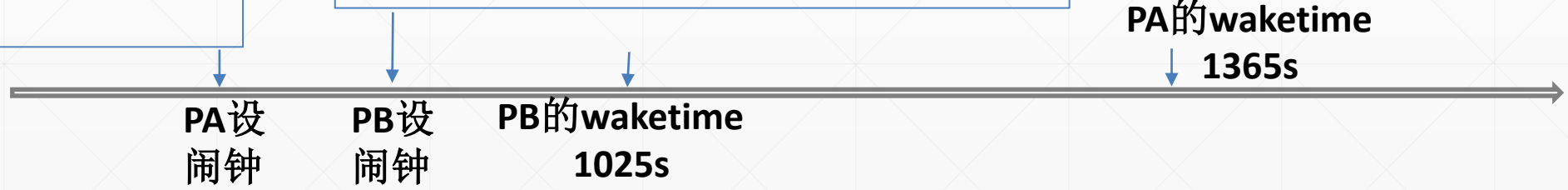
- T0时刻，发生时钟中断
  - 不是整数秒
  - 整数秒
    - 现运行进程用户态运行
    - 现运行进程核心态运行
- 整数秒，time值的修正会不会延迟？什么时候，time的值会修正回来？
- 整数秒现运行进程时间片用完，存在不马上放弃CPU的可能性吗？什么时候？

4、 时钟中断处理 和 Unix的定时器服务（闹钟）

- 系统调用　sleep(seconds)：应用程序执行sleep系统调用设置闹钟。执行它的进程会入睡，直至 time + seconds。之后进程恢复SRUN，接受CPU调度。**设置定时器的进程是执行系统调用sleep(seconds)入睡的进程。**
- time + seconds为进程的waketime。在所有waketime时刻，系统应该准确唤醒闹钟到时的那几个进程。

```
main()
{
    ……
    sleep(365);    // PA进程，1000s
    ……  // 365秒后再跑的代码；
}
```

```
main()
{
    ……
    sleep(5);      // PB进程于1020s设闹钟
    ……  // 5秒后再跑的代码；
}
```



实现

1、内核全局变量 tout，记录所有进程waketime的最小值
2、每个进程记录自己的waketime（自己的核心栈）
3、tout到期的时候，每个进程查看自己的waketime，到期Sleep系统调用返回；不到期，继续睡，重新设置tout的值

```
main()
{
   ……
   sleep(365);   // PA进程，1000s
   ……  // 365秒后再跑的代码；
}
```

sleep(seconds)系统调用:　　Sys_Sslep() 函数
1、waketime = time + seconds
2、当前没有进程设置定时器：　　Time::tout = wakeTime
　　有：　　tout = min(waketime，tout)
3、sleep(&Time::tout, 90) 入睡。设置 PCB（process[i]）：
　　p_stat = SWAIT　（阻塞，睡眠）
　　p_pri = 90　　（进程的优先级）
　　p_wchan = &tout（睡眠原因：闹钟）
　　swtch( ) // 保存现场，放弃CPU
4、waketime == tout ?
　　y: sleep系统调用返回，进程回用户态执行后续代码
　　n：goto 2

时钟中断处理程序clock()，每个整数秒
……
if (Time::time == Time::tout)
　　procMgr.WakeUpAll(&Time::tout);
……

```
void ProcessManager::WakeUpAll(unsigned long chan)
{
   遍历process数组，找到p_wchan = &tout的所有进程，
   对每个进程（process[i]）实施：
        p_stat = SRUN          runrun ++
        p_wchan = 0
}
```

代码

```
int SystemCall::Sys_Sslep()     // 系统调用sleep
{
        User& u = Kernel::Instance().GetUser();

        X86Assembly::CLI();
        unsigned int wakeTime = Time::time + u.u_arg[0];          // sleep(seconds)
        while( wakeTime > Time::time )                            // waketime没到，会再次入睡
        {
                if ( Time::tout <= Time::time || Time::tout > wakeTime )
                {
                        Time::tout = wakeTime;
                }
                u.u_procp->Sleep((unsigned long)&Time::tout, ProcessManager::PSLEP);
        }
        X86Assembly::STI();

        return 0;           /* GCC likes it ! */
}
```

sleep(seconds)系统调用Sslep函数的语义
1、waketime = time + seconds　　1365
2、tout = min(waketime，tout)　　1365
3、PCB（process[i]）：
　　p_stat = SWAIT　（阻塞，睡眠）
　　p_wchan = &tout（等着响闹钟）
4、swtch（保存现场后，放弃CPU）
5、waketime == tout ?
　　y：sleep系统调用返回，进程回用户态执行后续代码
　　n：goto 2

作业 1（写）

```
main()
{
    ……
    sleep(365);    // PA进程，1000s
    ……  // 365秒后再跑的代码；
}
```

```
main()
{
    ……
    sleep(5);      // PB进程于1020s设闹钟
    ……  // 5秒后再跑的代码；
}
```

分析时间轴上的4个时刻：tout值的变化，以及PA、PB进程控制块中发生变化的PCB属性

作业 2： sleep系统调用源代码分析 int SystemCall::Sys_Sslep()函数

作业 3： 系统有可能1025s的时刻无法唤醒 PB进程嘛？如果存在这种可能，PB进程唤醒时刻会延迟多久?

作业 4： 优化Unix系统的闹钟服务****

5、实现应用程序时间片轮转

```
class Process
{
public:
        ……
        int p_pri;              /* 进程动态优先数 */
        int p_cpu;          /* 衡量进程使用CPU的程度*/
        int p_nice;         /* 用于计算进程的静态优先数 */
        ……
}
```

UNIX进程的优先级动态变化，除非正在执行系统调用，否则

- p-pri = min {127，进程的静态优先数 + （p_cpu/16）}
- 进程的静态优先数 = PUSER(100) + p_nice

- p-pri = min {127 , 进程的静态优先数 + （p_cpu/16）}

```
void Process::SetPri()
{
        int priority;
        ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

        priority = this->p_cpu / 16;
        priority += ProcessManager::PUSER + this->p_nice;          //计算进程的动态优先数

        if ( priority > 255 )
        {
                priority = 255;
        }
        this->p_pri = priority;

        if ( priority > procMgr.CurPri )
        {
                procMgr.RunRun++;
        }
}
```

每次时钟中断，当前进程 p_cpu++

整数秒　1、所有进程p_cpu减SCHMAG。

2、重新计算所有用户态进程的优先数。

```
for( int i = 0; i < ProcessManager::NPROC; i++ )
{
        Process* pProcess = &procMgr.process[i];
        if ( pProcess->p_stat != Process::SNULL )
        {
                ......
                if ( pProcess->p_cpu > SCHMAG )
                {
                        pProcess->p_cpu -= SCHMAG;
                }
                else
                {
                        pProcess->p_cpu = 0;
                }
                if ( pProcess->p_pri > ProcessManager::PUSER )
                {
                        pProcess->SetPri();

                }
        }
}
```

```
void Process::SetPri()                          //    current->SetPri();
{
        int priority;
        ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

        priority = this->p_cpu / 16;
        priority += ProcessManager::PUSER + this->p_nice;          //计算进程的动态优先数

        if ( priority > 255 )
        {
                priority = 255;
        }
        this->p_pri = priority;

        if ( priority > procMgr.CurPri )        // 如果现运行进程连续使用CPU好久，RunRun++
        {
                procMgr.RunRun++;
        }
}
```

时钟中断 和 时间片轮转调度

每次时钟中断，当前进程 p_cpu++

整数秒　2、观察现运行进程在前一秒的行为。时间片用完，调度

```
/* 计算当前进程优先数 */
current->SetPri();
```

情景分析：
假设系统中存在4个用户态的进程 PA、PB、PC、PD，这些进程一直运算，不IO，不执行系统调用。进程的静态优先数相等：100，p_cpu是0。Process[5]、[7]、[8]、[9]分别是PA 、PB、PC和PD进程的PCB。T时刻是整数秒，PA先运行。观察这些进程如何轮流使用CPU。

- p-pri = min {127 ,进程的静态优先数 + （p_cpu/16）}

p_cpu

	T	T+1	T+2	T+3	T+4	T+5			
PA	0	40	20	0	0	40			
PB	0	0	40	20	0	0			
PC	0	0	0	40	20	0	……		
PD	0	0	0	0	40	20			

p_pri

	T	T+1	T+2	T+3	T+4	T+5			
PA	100	102	101	100	100	102			
PB	100	100	102	101	100	100			
PC	100	100	100	102	101	100	……		
PD	100	100	100	100	102	101			

SCHMAG = 20
HZ = 60

总结 1　时钟中断

- 维护时钟（wall clock)
- 记录每个进程的CPU使用程度，为系统提供精确的调度信息。这是实现分时操作系统的基础（不是之一，而是全部！）
- 时间片轮转调度（让应用程序轮流使用CPU)
- 响闹钟
- 其余内务处理
  - 进程图像在内存和磁盘之间搬迁
  - 当前进程信号处理
  - ……