

# 第八章 进程调度和时间

**UNIX**是分时分多进程操作系统，进程间的运行调度是整个系统运行控制的最根本内容。

## 进程调度的基本方式：

把每一次硬件时钟中断称为一个时钟“滴答”，由若干个时钟滴答构成一个时间片。

核心给每一个**用户**进程分配一个时间片，当该进程的时间片用完后，核心抢先该进程并调度另外一个进程运行。一段时间以后，核心又会重新调度该进程继续运行下去。以此方式，核心让各个进程轮流运行。

**核心**进程的运行：或者运行在不可被抢先的状态下；或者睡眠在某个中断级别上。

## 8.1 多级反馈循环调度算法

### 1、算法思想

核心给进程分配一个**CPU**时间片，抢先一个超过其时间片的进程，并把它反馈到若干优先级队列中的某一个队列上。

当进程的上下文切换结束时，核心执行**schedule\_process**算法来调度一个进程，即从处于“在内存中就绪（状态3）”和“被抢先（状态7）”状态的进程中，选取优先权最高的就绪进程。

如果若干个进程都具有相同的最高优先权，则核心选择在“就绪”状态时间最长的进程。

如果没有可运行的合格进程，核心则休闲等待，直到下次中断，下次中断最迟发生在下一个时钟滴答时。

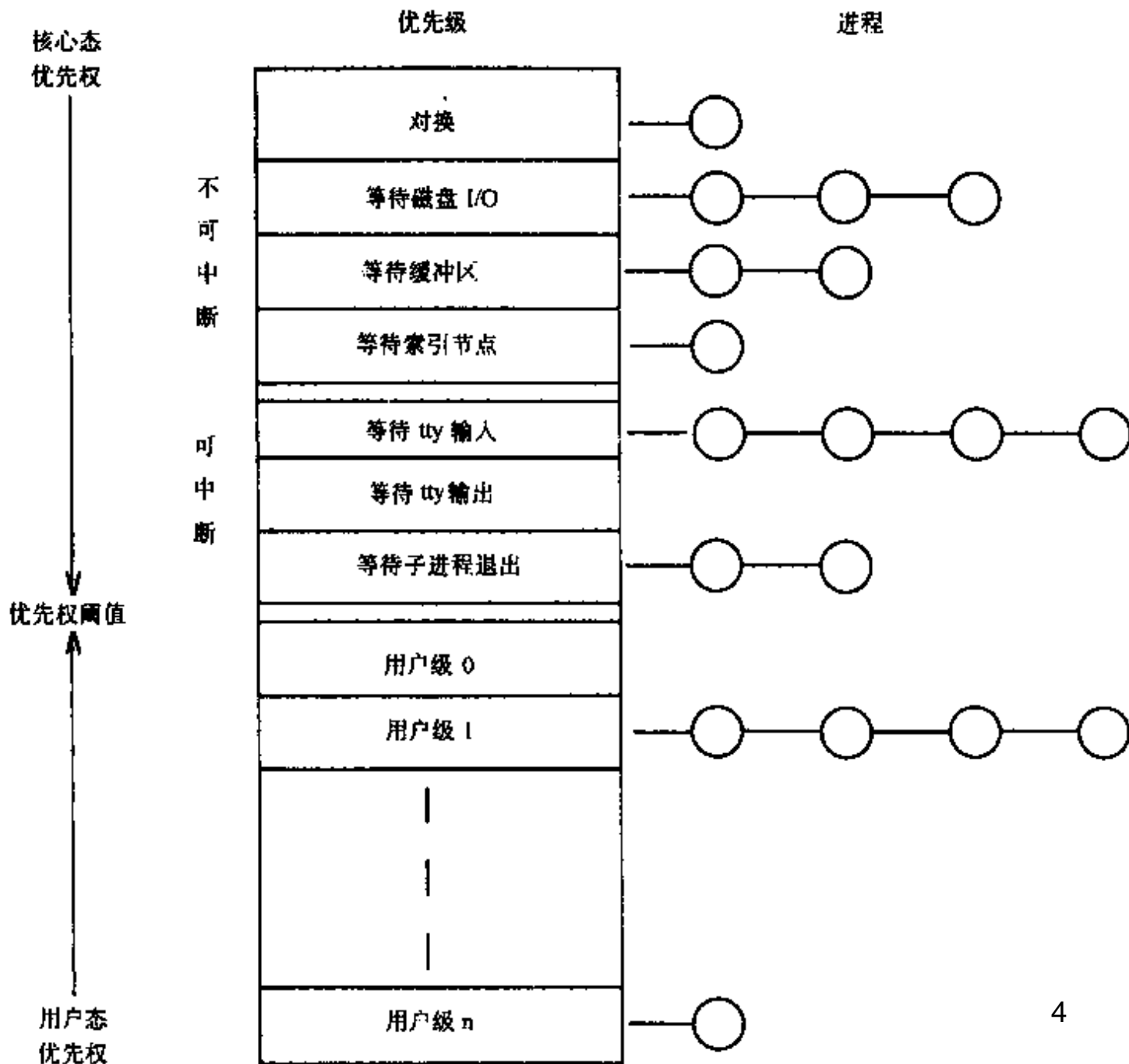
## 算法 `schedule_process`

输入：无

输出：无

```
{  
    while (没有能被选取运行的进程)  
    {  
        for (就绪队列中的每个进程)  
            取已装入内存的、优先级最高的进程;  
        if (没有合格运行的进程)  
            机器休闲;    /* 下次时钟中断使机器脱离休闲状态, 重新开始循环 */  
    }  
    将选取的进程从就绪队列中移出;  
    切换到被选取进程的上下文, 恢复其执行;  
}
```

## 2、调度参数



## 核心在三种情况下计算进程的优先权：

- ①、核心给一个即将进入睡眠态的进程赋予一个特定的优先权。  
—— 越是等待系统紧俏资源的进程，获得的优先权越高。
- ②、核心调整从核心态返回到用户态的进程的优先权。  
—— 进程在核心态下运行时拥有的较高优先权，必须在返回用户态时降低为用户级优先权。此外，该进程刚占用了宝贵的**CPU**时间，为公平起见也需降低本进程的优先权。
- ③、时钟中断处理程序每隔一个“时钟滴答”调整一次所有用户态进程的优先权。  
—— 核心运行调度算法，防止某个进程垄断**CPU**的使用。

在一个进程的时间片中，时钟可能要使它中断若干次——遇到多次“时钟滴答”，每次中断时，时钟中断处理程序都要重新计算所有进程（包括运行进程和等待进程）的**CPU**使用量，并由此调整各就绪进程的优先权值。

进程的**CPU**使用量：

$$\text{decay}(\text{CPU}) = \text{CPU}/2$$

其中的**CPU**是进程占用处理器的时间。

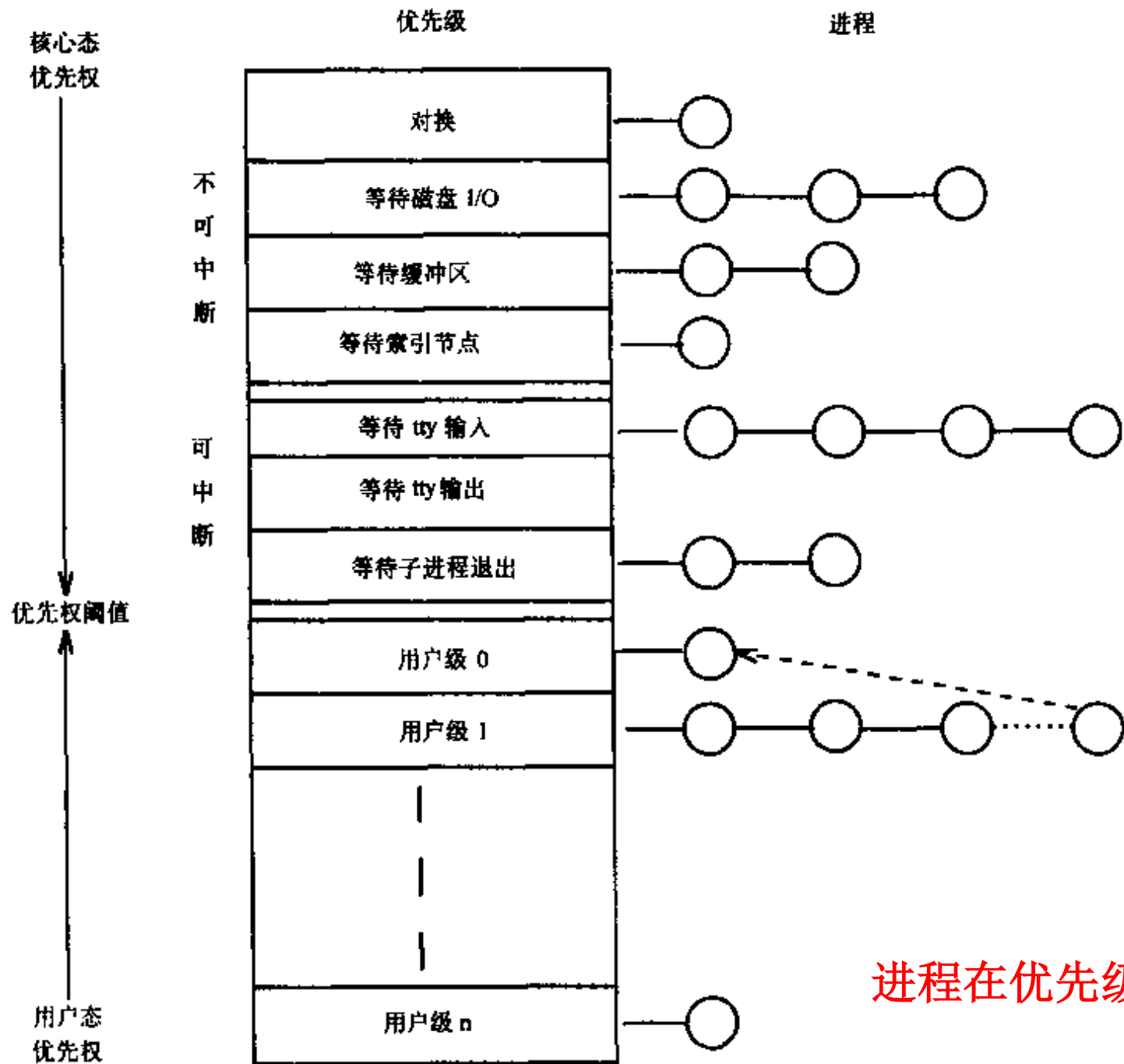
进程的优先权值（优先数）：

$$\text{priority} = (\text{CPU使用量}/2) + (\text{基级用户优先权值})$$

其中的“基级用户优先权值”就是**优先权阈值**。

优先权值越大，则优先级越低；优先权值越小，则优先级越高——运行进程的运行时间越长，优先权值越高，优先级降低；就绪进程等待的时间越长，优先权值越低，优先级升高。

**用户级优先权进程不能跨越阈值获得核心级优先权。**



## 说明：

1、如果进程正在执行一段临界区代码时收到时钟中断，它先“**记住**”此事，并继续运行，直到当前处理机执行级别降低之后的下一次时钟中断时，再重新计算进程优先权。

2、提高系统**实时性**（响应速度）的方法：

缩短时间片

提高衰减函数的衰减速度

——使进程轮转的频率更快，但系统开销更高。



### 3、进程调度实例：

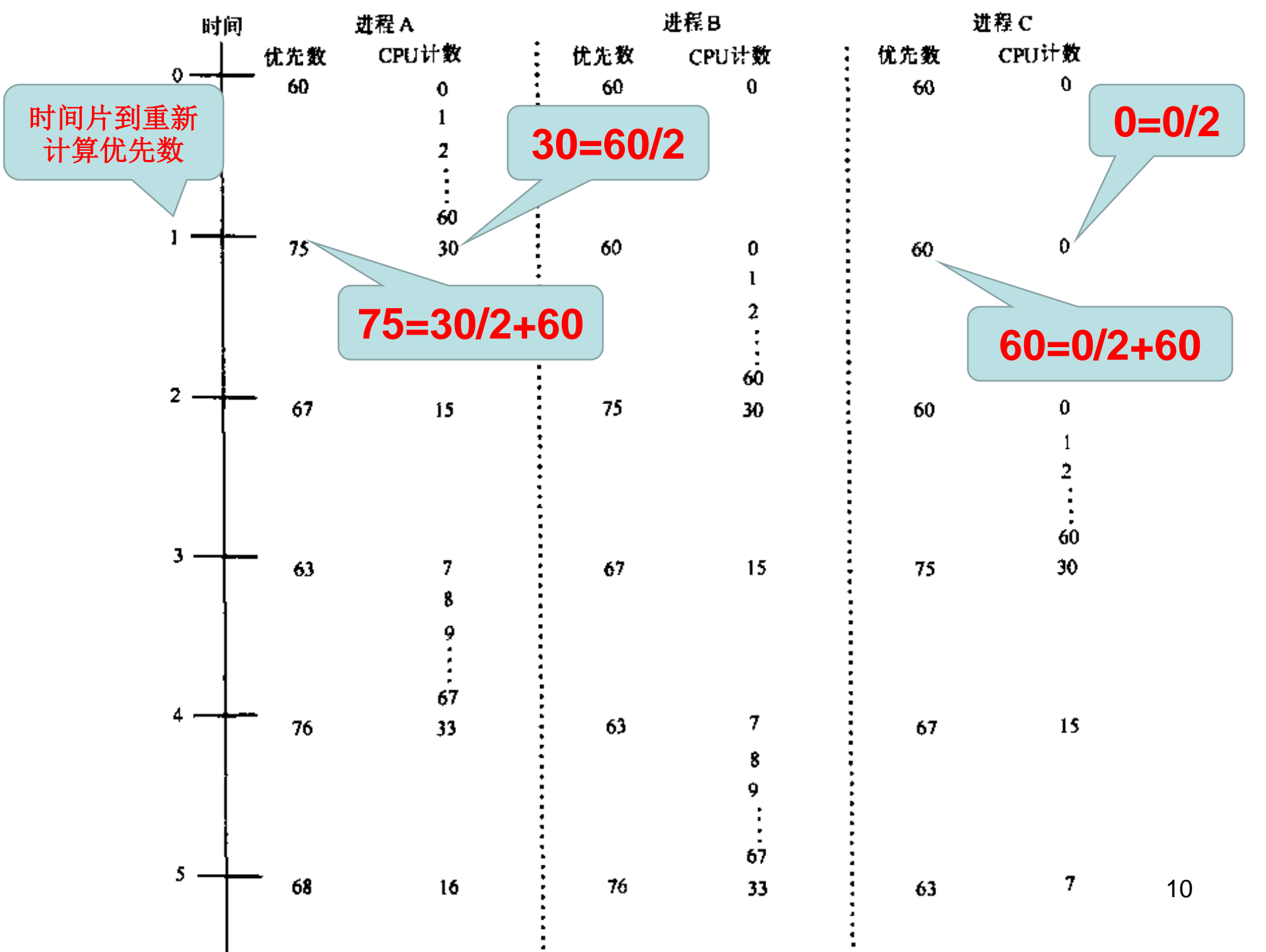
假设系统中有**A、B、C**三个同时建立的进程，具有相同的初始优先数**60**。系统中没有其它进程，这三个进程也没有做任何系统调用。时间片大小为一秒钟，每一秒钟产生**60**个时钟“滴答”，每个时间片到时，调用衰减函数重新计算每个进程的**CPU**使用量，必要的话就做上下文切换：

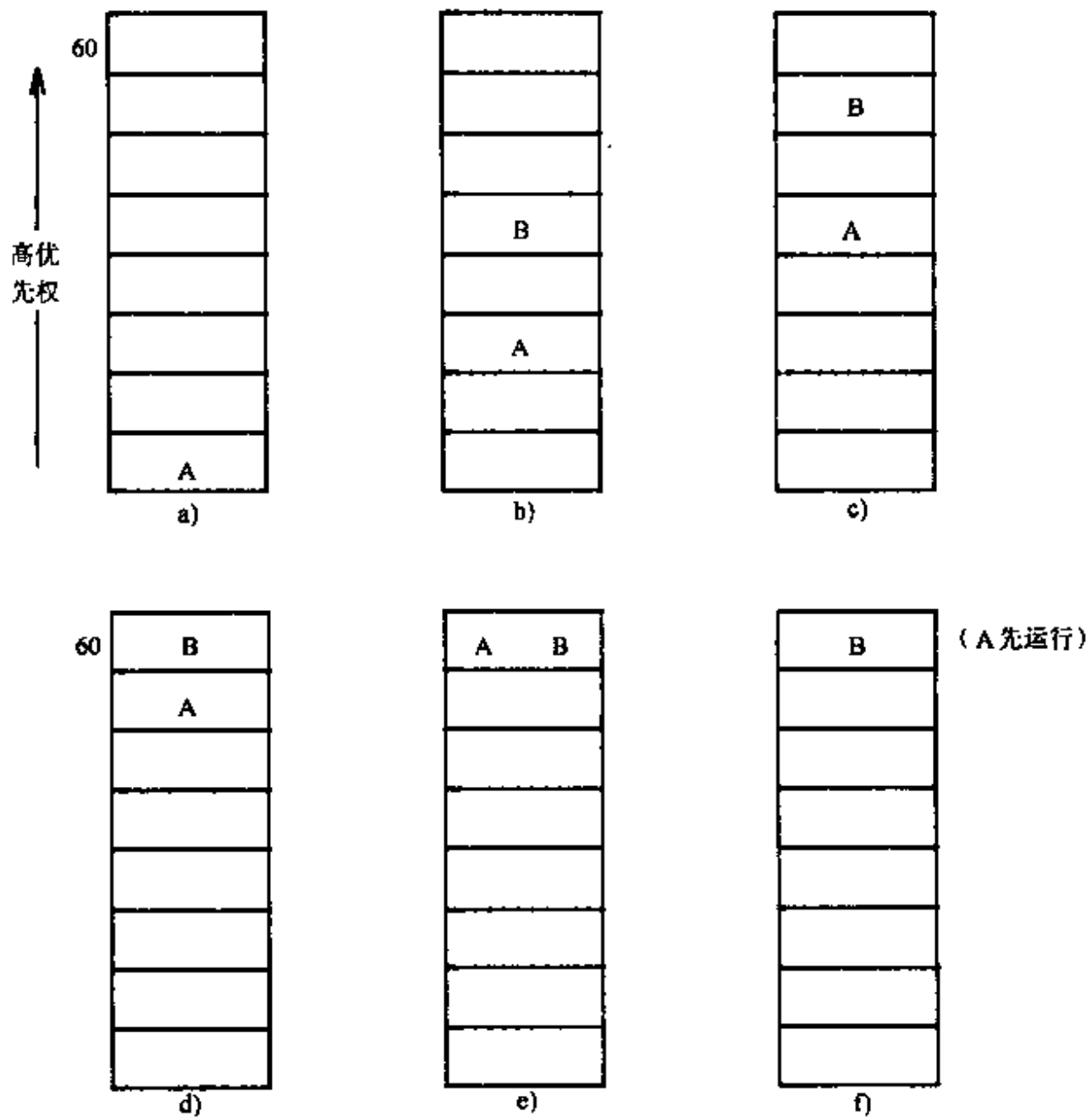
$$\text{CPU使用量} = \text{decay}(\text{时钟滴答数}) = \text{时钟滴答数}/2$$

则进程的优先数为：

$$\text{priority} = (\text{CPU使用量}/2) + 60$$

假设**A**进程先运行，下图为调度的顺序。





循环调度及相同优先权进程的调度原则

## 4、进程上下文切换的时机

①、当进程进入睡眠或退出（exit）时，它**必须**做上下文切换；

②、当进程从核心态返回到用户态时，它**有机会**做上下文切换。即是，当另一个具有较高优先权的进程就绪的话，核心就会抢先一个从核心态返回用户态的进程。出现这种抢先进程的原因有**两点**：

**a、**核心唤醒了一个比当前运行的进程具有更高优先权的进程。既然可以得到一个具有较高优先权的核心进程，那么当前运行的进程就不应该在用户态下继续运行。

**b、**当时钟中断处理程序修改了所有就绪进程的优先权时，发现当前运行进程已经用尽了时间片，而且许多进程的优先权已被修改到较高值，于是，核心重新调度一个进程。

## 5、调整进程优先数

系统调用 **nice** 用于调整一个进程的调度优先数：

**nice(value);**

**nice**的值**value**被加到计算进程优先数的公式中：

**priority=CPU使用量/2 + 基级用户优先权值 + value**

对于普通用户来讲，**value**必须是正整数，因此使用**nice**后必然使优先数**priority**的值加大，导致进程的优先级降低 —— 对其它进程来讲是有益无害的，因此非常“**nice**”。

在一些**UNIX**版本中**root**用户可以把**value**的值设置为**负值**，也就是超级用户可以提高一个进程的执行速度！

## 6、公平共享调度策略

问题：前述的调度算法不对用户做区分，进程间按“**绝对平均**”的原则分配时间片，无法满足现实应用中不同用户或不同进程需要不同响应级别的要求。

### 基本解决方案：

将用户分成若干个组 —— 公平共享组，系统将**CPU**时间平均分给各组，而不管各个组内有多少成员，组内成员再平均分配本组的**CPU**时间。

第一组  
25%CPU时间

A

A进程  
25%CPU时间

第二组  
25%CPU时间

C D

每个进程  
12.5%CPU时间

第三组  
25%CPU时间

E F G

每个进程  
8%CPU时间

第四组  
25%CPU时间

H I J K

每个进程  
6%CPU时间

## 6、公平共享调度策略

### 具体解决算法：

在前述的多级反馈循环调度算法的优先数计算公式中加入“**公平共享组优先数**”项，共享组中的任何进程使用**CPU**，则组中的其它进程的“**组优先数**”都要同时增加**CPU**计数值。

$$\text{priority} = \text{CPU使用量}/2 + \text{基级用户优先权值} + \text{组优先数}$$

在下图的实例中，**A**进程单独一组，**B**和**C**进程另成一组，两个组平等分享**CPU**的时间 —— 各组**50%**的**CPU**时间。

时间	进程 A			进程 B			进程 C		
	优先权值	CPU	组	优先权值	CPU	组	优先权值	CPU	组
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		...	...						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					...	...			...
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		...	...						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						...		...	...
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		...	...						
		78	78						
5	98	39	39	70	3	18	76	15	18



## 8.2 有关时间的系统调用

### 1、 **stime(pvalue);**

设置系统当前时间。超级用户调用**stime**来设定系统的当前时间，**pvalue**是以秒为单位的整数值，从**1970年1月1日零时零分**开始计时。系统命令**settime**主要就是调用**stime**来实现的。

### 2、 **time(tloc);**

获取系统当前时间。**tloc**用于存放返回给用户的时间值的单元，这个时间值同样是自**1970年1月1日零时零分**以来的秒数，应用程序需要将其转换为具体的年月日时分秒。命令**gettime**就主要是调用**time**。

## 8.2 有关时间的系统调用

### 3、 **times(tbuffer);** **struct tms \*tbuffer;**

读取本进程及其子进程的运行时间。其中**tbuffer**存放查询到的时间，由如下数据结构**tms**定义：

```
struct tms
{
    /* time_t是用于时间的数据结构 */
    time_t tms_utime;      /* 进程的用户态时间 */
    time_t tms_stime;      /* 进程的核心态时间 */
    time_t tms_cutime;     /* 子进程的用户态时间 */
    time_t tms_cstime;     /* 子进程的核心态时间 */
}
```

系统调用**times**的**返回值**是“从过去的任意一个时刻”开始所消逝的时间，通常是从系统初始的时间开始。

```

#include <sys/types.h>
#include <sys/times.h>
extern long times();
main()    /* 使用系统调用times的程序实例 */
{
    int t;
    struct tms pb1, pb2;    /* tms是有4个时间元素的数据结构 */
    long pt1, pt2;

    pt1 = times(&pb1);    /* 获取启动时间 */

    for ( i=0; i<10; i++)
        if (fork() == 0)
            child(i);

    for ( i=0; i<10; i++)
        wait((int *)0);    /* 等待10个子进程全部结束 */

    pt2 = times(&pb2);    /* 获取结束时间 */

    printf("parent real %u user %u sys %u cuser %u csys %u\n",
        pt2 - pt1, pb2.tms_utime - pb1.tms_utime, pb2.tms_stime - pb1.tms_stime,
        pb2.tms_cutime - pb1.tms_cutime, pb2.tms_cstime - pb1.tms_cstime);
}

```

```

child(int n)
{
    int i;
    struct tms cb1, cb2;
    long t1, t2;

    t1 = times(&cb1);

    for ( i=0; i<10000; i++)
        ;

    t2 = times(&cb2);

    printf("child %d: real %u user %u sys %u\n", n, t2 - t1,
           cb2.tms_utime - cb1.tms_utime, cb2.tms_stime - cb1.tms_stime);

    exit();
}

```

### 结论:

父进程的**用户时间**不等于子进程的用户时间之和  
 父进程的**系统时间**不等于子进程的系统时间之和

## 8.2 有关时间的系统调用

### 4、 **alarm(seconds);**

系统调用**alarm**用来设置闹钟软中断信号，其中**seconds**为秒数，用于设定从现在开始指定的时间后发出闹钟中断信号。

下图实例为一个无限循环，每分钟检查一次指定文件的存取时间，如果文件被访问过，则打印一个信息。

```

main(int argc, char *argv[ ])
{
    extern unsigned alarm( );
    extern wakeup( );
    struct stat statbuf;
    time_t axtime;
    axtime = (time_t)0;
    for ( ; ; )
    {
        /* 检查文件的存取时间 */
        if (stat(argv[1], &statbuf) == -1)
        {
            printf("file %s not there\n", argv[1]);
            exit( );
        }
        if (axtime != statbuf.st_atime)
        {
            printf("file %s accessed\n", argv[1]);
            axtime = statbuf.st_atime;
        }
        signal(SIGALRM, wakeup); /* 设置收到闹钟信号后如何处理 */
        alarm(60);
        pause( );      /* 睡眠等待软中断信号 */
    }
}

wakeup( )
{
}

```

## 8.3 时钟

### 时钟中断处理程序的功能：

- ① 重新启动时钟；
- ② 按内部定时器有计划地调用内部的核心函数；
- ③ 对核心进程和用户进程提供运行直方图分析的能力；
- ④ 收集系统和进程记账及统计信息；
- ⑤ 计时；
- ⑥ 在有请求时，向进程发送闹钟软中断信号；
- ⑦ 定时唤醒对换进程；
- ⑧ 控制进程调度；

算法 **clock**      /\* 时钟中断处理程序的算法 \*/

输入：无

输出：无

```
{
    重新启动时钟;    /* 为了获得下一次时钟中断信号 */
    if ( callout表非空)    /* 定时调用函数表 */
    {
        修改callout时间;
        如果时间已消逝, 安排调度callout函数;
    }
    if (核心直方图分析已开)
        记下中断时刻的程序计数器;
    if (用户直方图已开)
        记下中断时刻的程序计数器;
    收集系统统计信息;
    收集本进程统计信息;
    if (自上次执行此语句以来已经过了1秒钟或更多时间, 且中断不是发生在临界区代码区)
    {
        for (系统中的所有进程)
        {
            如果进程活动的话, 调整闹钟时间;
            修改CPU的使用量;
            if (进程在用户态执行)
                修改进程优先数;
        }
        必要的话, 唤醒对换进程
    }
}
```

过了一个时间片



## 8.3 时钟

### 1、重新启动时钟

系统时钟通常就是一个硬件计数器，对石英晶体的振动进行计数，当计数器达到指定值（如最大值，或进位位置一）时，产生一个硬件中断——时钟中断。操作系统常依次来同步和协调软件系统中各个程序（进程）的运行。

由于各个不同品牌的机器中，硬件计数器的差异巨大，以及对时钟系统准确性的要求，时钟中断处理程序通常使用汇编语言编写的。

每次时钟中断来临时，操作系统马上又立即启动时钟，以便获得下一次的时钟中断。

由于强调计时的及时和准确性，在**UNIX**中把时钟中断处理的优先设置得最高（除硬件故障外）。

## 8.3 时钟

### 2、系统的内部定时


在分时系统中的许多操作需要在实时的基础上调用核心函数来完成，如设备驱动和网络协议。

核心设置了一个callout表，其中含有当定时时间到时所要调用的函数名、函数参数、以时钟滴答为单位的定时时间。

用户不能直接控制callout表中的表项，这些表项是由核心在需要时用相关算法创建的。对callout表中的表项，核心不是按它们被放入表中的先后次序排序，而是按它们各自的“启动时间”进行排序。

在callout表中，各个表项的时间域记录的是前一表项启动后，到该表项被启动时的时间量。

# 例：在callout表中加入新表项f函数 —— 5个时钟滴答后调用f




函数名	启动时间
a( )	-2
b( )	3
c( )	10

加入f函数之前的callout表

3个时钟滴答后

13个时钟滴答后



函数名	启动时间
a( )	-2
b( )	3
f( )	2
c( )	8

加入f函数之后的callout表

3个时钟滴答后

5个时钟滴答后

13个时钟滴答后

创建一个新表项时，核心找出新表项的位置，并适当调整紧接新表项之后的那一项的时间域，而不需要改动其他表项的时间域！

## callout表项的调度流程:

- ① 时钟中断处理程序在每次时钟中断时，只把表中的第一项的时间域减1，后续的也就相应地自动减1了。
- ② 如果表中第一项的时间域小于或等于0，则应该调用该函数了。
- ③ 时钟中断处理程序并不直接调用该函数，而是产生一个较低级别的“软中断”——可编程中断来“记住”要调用该函数，开放级别较高的中断（如时钟中断等）。
- ④ 当所有较高级别的中断都处理完毕后，再运行相应的“软中断处理程序”。
- ⑤ 在callout表中应该调用某个函数的时刻到实际发生软中断之间，可能发生过包括时钟中断在内的多个中断，因此callout的第一个表项的时间域可能已被减为负值，软中断处理程序将清除已过时的callout表项，并调用相应的函数。
- ⑥ 由于callout表中前面几项可能为零或小于零，时钟中断处理程序必须找出第一个时间域为正值的表项，并使其减1。

## 8.3 时钟

### 3、直方图分析

核心直方图驱动程序在时钟中断时，对系统的活动（地址）进行采样，以便监视系统在核心态和用户态下的执行时间的相对比例，对系统性能进行评估。

直方图分析程序有一个用于采样的核心地址表，表中包含有核心函数的地址。

允许核心进行直方图分析时，时钟中断处理程序就调用直方图驱动程序对应的中断处理程序，对当前程序计数器的取值进行记录，并与核心地址表相比较，以确定当前正在运行哪个程序。

## 示例：核心算法的采样地址表

算法	地址	计数
bread	100	5
breada	150	0
bwrite	200	0
breise	300	2
getblk	400	1
user	—	2

**5次在地址100~149之间**

**0次在地址150~199之间**

**0次在地址200~299之间**

**2次在地址300~399之间**

**1次在地址400以上**

**2次在用户地址空间**