# 第3章 进程同步与通信

操作系统中的并发进程有些是独立的，有些需要相互协作。独立的进程在系统中执行时不影响其他进程，也不被其他进程影响；而另一些进程需要与其他进程共享数据，以完成一项共同的任务，这些进程之间具有协作关系。如果对协作进程的活动不加约束，就会使系统混乱。如，当多个进程争用一台打印机时，有可能多个进程的输出结果交织在一起，难以区分。所以，为了保证系统的正常活动，使程序的执行具有可再现性，操作系统必须提供某种机制。

进程之间的协作关系包括互斥、同步和通信。

互斥是指多个进程不能同时使用同一个资源，当某个进程使用某种资源时，其他进程必须等待。

同步是指多个进程中发生的事件存在着某种时序关系，某些进程的执行必须先于另一些进程。

进程通信是指多个进程之间要传递一定量的信息。

## 3.1 进程同步与互斥

### 3.1.1 *并发原理*

**1. 并发带来的问题**

在单处理机多道程序环境系统中，进程被交替执行，表现出一种并发执行的特征，即使不能实现真正的并行处理，进程间来回切换需要一定的开销，这种交替执行在处理效率上还是带来了很大的好处。但是，由于并发执行的进程之间相对执行速度是不可预测的，它取决于其他进程的活动、操作系统的调度策略等。这就带来了以下困难。

(1) 全局变量的共享充满了危险。如果两个进程都使用同一个全局变量，并且都对该变量进行读写操作，那么不同的读写执行顺序是非常关键的。

(2) 操作系统很难最佳地管理资源的分配。如果某个进程请求使用某个特定的 I/O 设备，并得到了这个设备，但该进程在使用该设备前被挂起了，操作系统仍然把这个设备锁定给该进程，而不能分配给其他进程，因为操作系统不知道被挂起的进程何时又将执行。此外，资源分配还会导致死锁的危险。

(3) 定位程序的错误是很困难的。这是因为并发程序存在不确定性和不可再现性。

因此“并发”给操作系统的设计和管理带来了很多问题，操作系统为此要关注的事情有以下几方面。

(1) 操作系统必须记录每个进程的情况，并通过进程控制块实现。

(2) 操作系统必须为每个进程分配和释放各种资源，这些资源包括处理机、存储器、文件和 I/O 设备。

(3) 操作系统必须保护每个进程的数据和资源，避免遭到其他进程的干涉和破坏。

(4) 保证进程执行结果的正确性，进程的执行结果与速度无关。

以上 4 个问题中，第(1)个问题已经在第 2 章内容中解决，第(2)、(3)个问题涉及存储管理、文件管理和设备管理相关的技术，本节要重点解决的是第(4)个问题。

**2. 进程的交互**

按进程之间是否知道对方的存在以及进程的交互方式划分，进程的交互可以分为以下 3 种情况。

(1) 进程之间不知道对方的存在。这是一些独立的进程，它们不会一起工作。只是无意地同时存在着。尽管这些进程不一起工作，但是操作系统需要知道它们对资源的竞争情况。例如，两个无关的进程都要使用同一磁盘文件或打印机，操作系统必须控制和管理对它们的访问。

(2) 进程间接知道对方。进程并不需要知道对方的进程标识符，但它们共享某些数据，它们在共享数据时要进行合作。

(3) 进程直接得知对方。进程通过进程标识符互相通信，用于合作完成某些任务。

表 3-1 列出了 3 种可能的认知程度和结果，但实际情况有时并不像表中给出的那么清晰，几个进程可能既要竞争，又要合作，操作系统需要检查进程之间的确切关系，并为它们服务。

**表 3-1　进程的交互**

| 知 道 程 度 | 关系 | 对其他进程的影响 | 潜在的控制问题 |
|---|---|---|---|
| 进程间不知道对方 | 竞争 | 进程的执行结果与其他进程无关 | 互斥<br>死锁<br>饿死 |
| 进程间接知道对方 | 共享合作 | 进程的执行结果可能依赖于从其他进程中得到的消息 | 互斥<br>死锁<br>饿死<br>数据一致性 |
| 进程直接得知对方 | 通信合作 | 进程的执行结果可能依赖于从其他进程中得到的消息 | 死锁<br>饿死 |

进程的并发执行使进程之间存在着交互，进程间的交互关系包括互斥、同步和通信。

进程互斥是指由于共享资源所要求的排他性，进程之间要相互竞争，某个进程使用这种资源时，其他进程必须等待。换句话说，互斥是指多个进程不能同时使用同一个资源。这种情况下，进程之间知道对方的程度最低。

进程同步是指多个进程中发生的事件存在着某种时序关系，必须协同动作，相互配合，以共同完成一个任务。进程同步的主要任务是使并发执行的诸进程有效地共享资源和相互合作，从而使程序的执行具有可再现性。这种情况比进程之间的互斥知道对方的程度要高，因为进程之间要合作。

进程通信是指多进程之间要传递一定的信息。这种情况下,进程之间知道对方的程度最高,需要传递的信息量也最大。

**3. 进程互斥**

在日常生活中,人与人之间会竞争某一事物,如交叉路口争抢车道、篮球比赛中争抢篮板球。在计算机系统中,进程之间也存在这种竞争,如两个进程争抢一台打印机。对于这种竞争问题,最简单的解决办法就是先来先得,具体地说,在交叉路口,先到者先通过,后到者必须等待先到者通过后再通过;在篮球比赛中,先抢到篮板球者得球;在计算机系统中也一样,先申请打印机的一方先使用打印机,等它用完后才可给其他进程使用。在一个进程使用打印机期间,其他进程对打印机的使用申请不予满足,这些进程必须等待。

综上可以看出,竞争双方本来毫无关系,但由于竞争同一资源,使两者产生了相互制约的关系,这种制约关系就是互斥。所谓互斥就是指多个进程不能同时使用同一资源。

**4. 进程同步**

在4×100米接力赛中,运动员之间要默契配合,在接棒区,前一棒运动员要把棒交给下一棒的运动员,4个运动员密切配合才能完成比赛。在工厂的流水线上,每道工序都有自己特定的任务,前一道工序没有完成或完成的质量不合格,后一道工序就不能继续进行。运动员之间和工序之间的这种关系就是一种同步关系。日常生活中的这种同步关系在计算机的进程之间同样存在。例如A、B、C 3个进程,A进程负责输入数据,B进程负责处理数据,C进程负责输出数据,这3个进程之间就存在着同步关系,即A必须先执行,B次之,C最后执行,否则不能得到正确的结果。

通过以上分析可以看出,所谓进程同步,是指多个进程中发生的事件存在着某种时序关系,它们必须按规定时序执行,以共同完成一项任务。

## 3.1.2 临界资源与临界区

**1. 临界区与临界资源的概念**

在计算机中,有些资源允许多个进程同时使用,如磁盘;而另一些资源只能允许一个进程使用,如打印机、共享变量。如果多个进程同时使用这类资源,就会引起激烈的竞争。操作系统必须保护这些资源,以防止两个或两个以上的进程同时访问它们。那些在某段时间内只允许一个进程使用的资源称为临界资源(Critical Resource),每个进程中访问临界资源的那段程序称为临界区(Critical Section)。

几个进程共享同一临界资源,它们必须以互相排斥的方式使用临界资源,即当一个进程正在使用临界资源且尚未使用完毕时,其他进程必须延迟对该资源的进一步操作,在当前进程使用完毕之前,不能从中插入使用这个临界资源,否则将会造成信息混乱和操作出错。

例如$P_1$、$P_2$两进程共享变量COUNT(COUNT的初值为5),$P_1$、$P_2$两个程序段如下。

```
P1:                          P2:
{                            {
  R1 = COUNT;                  R2 = COUNT;
  R1 = R1 + 1;                 R2 = R2 + 1;
  COUNT = R1;                  COUNT = R2;
}                            }
```

分析以上两个进程的执行可能会出现以下几种情况。

(1) 进程的执行顺序 $P_2 \rightarrow P_1$，即 $P_2$ 执行完毕后，$P_1$ 再执行。此时的执行结果为 $P_2$ 执行完毕，COUNT 为 6；$P_1$ 执行完毕，COUNT 为 7。

(2) 两个进程交替执行，具体为进程 $P_1$ 执行{$R_1$ = COUNT}后进程 $P_2$ 执行{$R_2$ = COUNT}，然后进程 $P_1$ 再执行{$R_1 = R_1 + 1$；COUNT = $R_1$}，最后进程 $P_2$ 执行{$R_2 = R_2 + 1$；COUNT = $R_2$}。执行结果为进程 $P_1$ 所有程序段执行完毕后 COUNT 为 6，进程 $P_2$ 所有程序段执行完毕后 COUNT 为 6。

以上两种执行顺序产生了两个不同的执行结果。

**2. 进程访问临界区的一般结构**

用 Bernstein 条件考察以上两个进程。

$P_1$ 的读集和写集分别是 $R(P_1)$={$R_1$，COUNT}、$W(P_1)$={$R_1$，COUNT}；$P_2$ 的读集和写集分别是 $R(P_2)$={$R_2$，COUNT}、$W(P_2)$={$R_2$，COUNT}。

而 $R(P_1) \cap W(P_2) \neq \{\}$不符合 Bernstein 条件，因此，必须对进程 $P_1$ 和 $P_2$ 的执行施加某种限制，否则 $P_1$ 和 $P_2$ 将无法并发执行。也就是说，$P_1$ 和 $P_2$ 两个进程在执行时必须等一个进程执行完毕，另一个进程才可以执行。在这里，变量 COUNT 是一个临界资源，$P_1$ 和 $P_2$ 的两个程序段是临界区。

可见，不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它们进行访问。

显然，若能保证诸进程互斥地进入临界区，就可实现它们对临界资源的互斥访问。为此，每个进程在进入临界区之前应对要访问的临界资源进行检查，看它是否正在被访问。如果此刻临界资源未被访问，进程便可以进入临界区，对资源进行访问，并设置它正被访问的标志；如果此刻临界资源正被某进程访问，则进程不能进入临界区。因此，必须在临界区前面增加一段用于进行上述检查的代码，这段代码称为进入区(Enter Section)。相应地，在临界区后面也要加上一段称为退出区(Exit Section)的代码，用于将临界区正被访问的标志恢复为未被访问标志。进程中除去上述进入区、临界区及退出区之外的其他部分的代码称为剩余区(Remainder Section)。图 3-1 所示为进程访问临界区的一般结构。

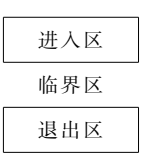


图 3-1 进程访问临界区的一般结构

**3. 临界区进入准则**

为了实现进程互斥，可用软件或硬件的方法在系统中设置专门的同步机制来协调多个进程，但所有同步机制都必须遵循下述 4 个准则。

(1) 空闲让进。当无进程处于临界区时，临界资源处于空闲状态，允许进程进入临界区。

(2) 忙则等待。当已有进程进入临界区时，临界资源正在被访问，其他想进入临界区的进程必须等待。

(3) 有限等待。对于要求访问临界资源的进程，应保证在有效的时间内进入，以免进入“死等”状态。

(4) 让权等待。当进程不能进入临界区时，应立即释放处理机，以免其他进程进入“忙等”状态。

### 3.1.3 互斥实现的硬件方法

为了解决进程互斥进入临界区的问题,需要采取有效措施。利用硬件实现互斥的方法有禁止中断和专用机器指令两种方法。

**1. 禁止中断**

在单处理机环境中,并发执行的进程不能在CPU上同时执行,只能交替执行。另外,对一个进程而言,它将一直运行,直到被中断。因此,为了保证互斥,只要保证一个进程不被中断就可以了,这可以通过系统内核开启、禁止中断来实现。

进程可以通过图3-2所示的方法实现互斥。

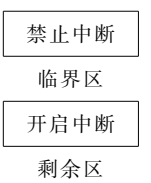


图3-2 用禁止中断的方法访问临界区

由于在临界区内进程不能被中断,故保证了互斥。但该方法的代价很高,进程被限制只能交替执行。

另外,在多处理机环境中,禁止中断仅对执行本指令的CPU起作用,对其他CPU不起作用,也就不能保证对临界区的互斥进入。

**2. 专用机器指令**

在很多计算机(特别是多处理机)中设有专用指令来解决互斥问题。依据所采用指令的不同,硬件方法分成TS指令和Swap指令两种。

1) TS(Test and Set)指令

TS指令的功能是读出指定标志后把该标志设为true,TS指令的功能可以用如下函数来描述。

```
boolean  TS(lock);
boolean  lock;
{    boolean  temp;
temp = lock;
lock = true;
return  temp;
}
```


图3-3 用TS指令访问临界区

为了实现进程对临界区的访问,可为每个临界资源设置一个布尔变量lock,表示资源的两种状态,true表示正被占用,false表示空闲。在进入区检查和修改标志lock;有进程在临界区时,循环检查,直到其他进程退出时通过检查进入临界区。所有要访问临界资源的进程在进入区和退出区的代码是相同的,如图3-3所示。

2) Swap指令

Swap指令的功能是交换两个字节的内容,可以用如下函数描述Swap指令。

```
void  Swap(a,b);
boolean  a,b;
{    boolean  temp;
```

```
temp = a;
a = b;
b = temp;
}
```

利用 Swap 指令实现进程互斥算法，为每个临界资源设置一个全局布尔变量 lock，初始值为 false；每个进程设置一个局部布尔变量 key。在进入区利用 Swap 指令交换 lock 与 key 的内容，然后检查 key 的状态；有进程在临界区时，循环交换和检查过程，直到其他进程退出时检查通过，进入临界区，如图 3-4 所示。

```
key=true;
do
{
  Swap(lock,key);
}while (key);
```

临界区；

```
lock=false;
```

剩余区；

图 3-4　用 Swap 指令访问临界区

3）硬件方法的优点

硬件方法由于采用硬件处理器指令能很好地把修改和检查操作结合在一起而具有明显的优点。具体地说，硬件方法的优点有以下几点。

（1）适用范围广。硬件方法适用于任意数目的进程，单处理机和多处理机环境都能用。

（2）简单。硬件方法的标志设置简单，容易验证其正确性。

（3）支持多个临界区。在一个进程中有多个临界区，只需为每个临界区设置一个布尔变量。

4）硬件方法的缺点

硬件方法也有无法克服的缺点，主要包括以下两方面。

（1）进程在等待进入临界区时，不能做到“让权等待”。

（2）由于进入临界区的进程是从等待进程中随机选择的，可能造成某个进程长时间不能被选上，从而导致“饥饿”现象。

### 3.1.4　互斥实现的软件方法

有许多方法可以实现互斥。第一种方法是让希望并发执行的进程自己来完成。不论是系统程序还是应用程序，当需要与另一个进程互斥时，不需要操作系统提供任何支持，自己通过软件来完成。尽管该方法已经被证明会增加许多处理开销和错误，但通过分析这种方法，可以更好地理解并发处理的复杂性。第二种方法是使用专门的机器指令来完成，这种方法的优点是可以减少开销，但与具体的硬件系统相关，很难成为一种通用的解决方案。第三种方法是由操作系统提供某种支持。

通过平等协商方式实现进程互斥的最初方法是软件的方法。其基本思路是在进入区检查和设置一些标志，如果已有进程在临界区，则在进入区通过循环检查进行等待；在退出区修改标志。

**1. 算法 1：单标志算法**

假如有两个进程 $P_0$、$P_1$ 要互斥地进入临界区，设置公共整型变量 turn，用于指示进入临界区的进程标识，进程在进入区通过循环检查变量 turn 确定是否可以进入，即当 turn 为 0 时，进程 $P_0$ 可进入，否则循环检查该变量，直到 turn 变为 0 为止。在退出区将 turn 改成另一个进程的标识，即 turn=1，从而使 $P_0$、$P_1$ 轮流访问临界资源，如图 3-5 所示。

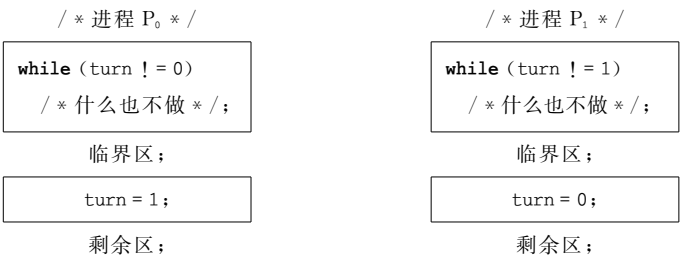


图 3-5　单标志算法

该算法可以保证任何时刻至多只有一个进程进入临界区,但它的缺点是强制性轮流进入临界区,不能保证“空闲让进”。

**2. 算法 2:双标志、先检查算法**

为了克服算法 1 强制性轮流进入临界区的缺点,可以考虑修改临界区标志的设置。设标志数组 flag[2],初始时设每个元素为 false,表示所有进程都未进入临界区,若 flag[0]=true,则表示进程 $P_0$ 进入临界区执行。

每个进程进入临界区时,先查看临界资源是否被使用,若正在使用,该进程等待,否则进入,从而解决了“空闲让进”问题。

图 3-6 所示是两个进程的代码。进程 $P_0$ 的代码中,程序先检查进程 $P_1$ 是否在临界区,若 $P_1$ 没有在临界区,则修改标志 flag,进程 $P_0$ 进入临界区。

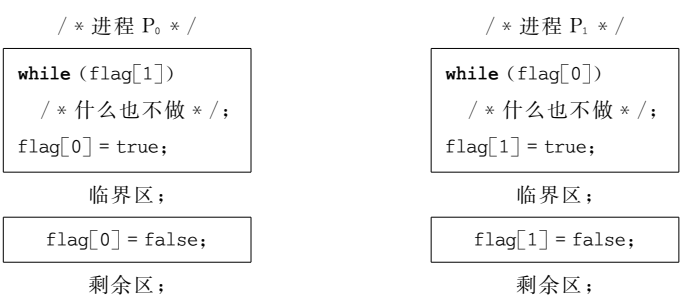


图 3-6　双标志、先检查算法

该算法解决了“空闲让进”的问题,但如果 $P_0$ 和 $P_1$ 几乎同时要求进入临界区,因都发现对方的访问标志 flag 为 false,于是两进程都先后进入临界区,所以该算法又出现了可能同时让两个进程进入临界区的缺点,不能保证“忙则等待”。

**3. 算法 3:双标志、先修改后检查算法**

算法 2 的问题是,当进程 $P_0$ 观察到进程 $P_1$ 的标志为 false 后,便将自己的标志 flag 改为 true,这需要极短的一段时间,而正是在此期间,进程 $P_1$ 观察进程 $P_0$ 的标志为 false,而进入临界区,因而造成两个进程同时进入的问题。

解决该问题的方法是先修改后检查,这时标志 flag 的含义是进程想进入临界区,如图 3-7 所示。

该算法可防止两个进程同时进入临界区,但它的缺点是可能两个进程因过分“谦让”而都进不了临界区。

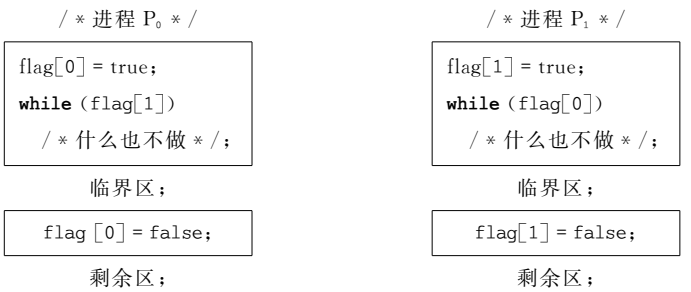


图 3-7　双标志、先修改后检查算法

**4. 算法 4：先修改、后检查、后修改者等待算法**

结合算法 1 和算法 3 的概念，标志 flag[0]为 true 表示进程 $P_0$ 想进入临界区，标志 turn 表示要在进入区等待的进程标识。在进入区先修改后检查，通过修改同一标志 turn 来描述标志修改的先后；检查对方标志 flag，如果对方不想进入，自己再进入。如果对方想进入，则检查标志 turn，由于 turn 中保存的是较晚的一次赋值，因此较晚修改标志的进程等待，较早修改标志的进程进入临界区，如图 3-8 所示。

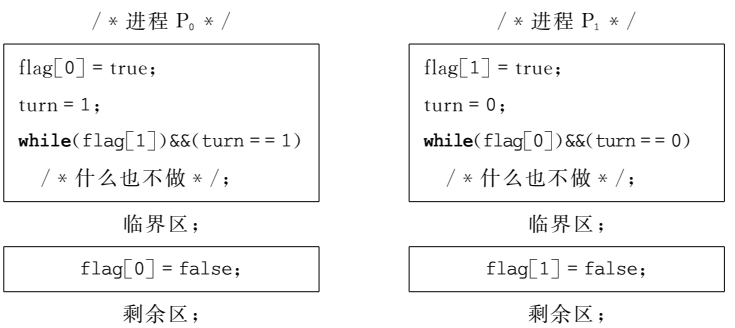


图 3-8　先修改、后检查、后修改者等待算法

至此，算法 4 可以正常工作，即实现了同步机制中的前两条——“空闲让进”和“忙则等待”。

但从以上软件方法中可以发现，对于 3 个以上进程的互斥又要区别对待。因此用软件方法解决进程互斥的问题有一定难度，且有很大的局限性，因而现在很少有人用这样的方法。

## 3.1.5　信号量和 PV 操作

1965 年，荷兰学者 Dijkstra 提出的信号量机制是一种卓有成效的解决进程同步问题的工具。该机制提出后得到了长期且广泛的应用，并得到了很大的发展。

**1. 信号量的定义**

Dijkstra 最初定义的信号量包括一个整型值 s 和一个等待队列 s. queue，信号量只能通过两个原语 P、V 操作来访问它，信号量的定义如下。

```
struct  semaphore{
  int  value;
  struct  PCB * queue;
}
```

P原语所执行的操作可用如下函数wait(s)来表示。

```
void  wait(semaphore  s)
{  s.value = s.value - 1;
   if  (s.value<0)
       block(s.queue);          /* 将进程阻塞,并将其投入等待队列 s.queue */
}
```

V原语所执行的操作可用下面的函数signal(s)来表示:

```
void  signal(semaphore  s)
{  s.value = s.value + 1;
   if  (s.value<=0)
       Wackup(s.queue);
/* 唤醒阻塞进程,将其从等待队列 s.queue 取出,投入就绪队列 */
}
```

**2. 信号量的物理意义**

(1) 在信号量机制中,信号量的初值s.value表示系统中某种资源的数目,因而又称为资源信号量。

(2) P操作意味着进程请求一个资源,因此描述为s.value=s.value−1;当s.value<0时,表示资源已经分配完毕,因而进程所申请的资源不能够满足,进程无法继续执行,所以进程执行block(s.queue)自我阻塞,放弃处理机,并插入等待该信号量的等待队列。

(3) V操作意味着进程释放一个资源,因此描述为s.value=s.value+1;当s.value≤0时,表示在该信号量的等待队列中有等待该资源的进程被阻塞,故应调用wakeup(s.queue)原语将等待队列中的一个进程换醒。

(4) 当s.value<0时,|s.value|表示等待队列的进程数。

**3. 用信号量解决互斥问题**

如果信号量的初值为1,表示仅允许一个进程访问临界区,此时的信号量转换为互斥信号量。P操作和V操作分别置于进入区和退出区,如定义mutex为互斥信号量,其初值为1,P、V操作的位置如图3-9所示。

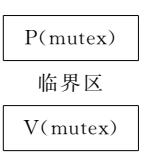


图3-9 用信号量解决互斥问题

例如,对于前文举例的$P_1$、$P_2$两进程共享全程变量COUNT(COUNT的初值为5)的问题,用信号量来解决$P_1$、$P_2$两个程序段如下。

```
semaphore mutex = 1;
P1:                                  P2:
{                                    {
  P(mutex);                            P(mutex);
  R1 = COUNT;                          R2 = COUNT;
  R1 = R1 + 1;                         R2 = R2 + 1;
  COUNT = R1;                          COUNT = R2;
  V(mutex);                            V(mutex);
}                                    }
```

如此，设置了信号量之后，无论 $P_1$、$P_2$ 两进程按照怎样的次序执行，其结果都是一样的，即 COUNT 最终的值为 7。

**4. 用信号量解决同步问题**

利用信号量可以实现进程之间的同步，即可以控制进程执行的先后次序。如果有两个进程 $P_1$ 和 $P_2$，要求 $P_2$ 必须在 $P_1$ 执行完毕之后才可以执行，则只需要设置一个信号量 S，其初值为 0，将 V(S)操作放在进程 $P_1$ 的代码段 $C_1$ 后面，将 P(S)操作放在进程 $P_2$ 的代码段 $C_2$ 前面，代码所示如下。

```
/* 进程 P1 */          /* 进程 P2 */
C1;                    P(S);
V(S);                  C2;
```

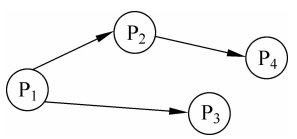


图 3-10　4 个进程之间的执行次序

如图 3-10 所示，进程关系中有 4 个并发执行的进程 $P_1$、$P_2$、$P_3$ 和 $P_4$，它们之间的关系是 $P_1$ 首先被执行；$P_1$ 执行完毕 $P_2$、$P_3$ 才执行；而 $P_4$ 只有在 $P_2$ 执行完毕后才能执行。为了实现它们之间的同步关系，可以写出如下并发程序。

```
semaphore s1 = s2 = s3 = 0;

/* 进程 P1 */      /* 进程 P2 */      /* 进程 P3 */      /* 进程 P4 */
{                  {                  {                  {
C1;                P(s1);             P(s2)              P(s3);
V(s1);             C2;                C3;                C4;
V(s2);             V(s3);             }                  }
}                  }
```

## 3.2　经典进程同步与互斥问题

在多道程序设计环境中，进程同步是一个非常重要的问题，下面讨论几个经典的进程同步问题。从中可以看出，对于信号量的使用，主要是如何选择信号量和如何安排 P、V 操作在程序中的位置。

### 3.2.1　生产者—消费者问题

**1. 问题描述**

生产者—消费者问题是指有两组进程共享一个环形的缓冲池（见图 3-11），一组进程称为生产者，另一组进程称为消费者。缓冲池是由若干个大小相等的缓冲区组成的，每个缓冲区可以容纳一个产品。生产者进程不断将生产的产品放入缓冲池，消费者进程不断将产品从缓冲池中取出。指针 $i$ 和 $j$ 分别指向当前的第一个空闲缓冲区和第一个存满产品的缓冲区（斜线部分）。

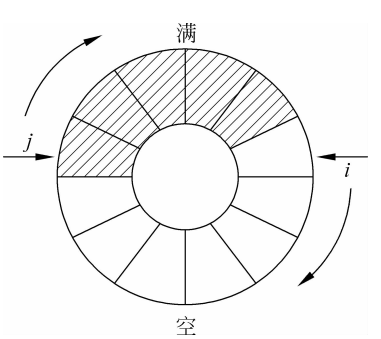


图 3-11　环形缓冲池

**2. 用信号量解决生产者—消费者问题**

在生产者—消费者问题中,既存在着进程同步问题,也存在着临界区的互斥问题。当缓冲区满时,表示供大于求,生产者必须停止生产,进入等待状态,同时唤醒消费者;当所有缓冲区都空时,表示供不应求,消费者必须停止消费,唤醒生产者。这就是生产者进程和消费者进程的同步关系。

对于缓冲池,它显然是一个临界资源,所有生产者和消费者都要使用它,而且都要改变它的状态,故对于缓冲池的操作必须是互斥的。

下面是用信号量及P、V操作解决生产者—消费者问题的形式化描述。

```
semaphore  mutex = 1;
semaphore  empty = n;
semaphore  full = 0;
int  i,j;
ITEM  buffer[n];
ITEM  data_p,data_c;

void  producer()           /* 生产者进程 */
{
while  (true)
{
    produce an item in data_p;
    P(empty);
    P(mutex);
    buffer[i] = data_p;
    i = (i + 1) % n;
    V(mutex);
    V(full);
  }
}
void  consumer()           /* 消费者进程 */
{
while  (true)
{
    P(full);
    P(mutex);
    data_c = buffer[j];
    j = (j + 1) % n;
    V(mutex);
    V(empty);
    consume the item in data_c;
  }
}
```

**3. 要注意的问题**

在生产者—消费者问题中要注意以下几个问题。

(1) 把共享缓冲池中的 $n$ 个缓冲区视为临界资源，进程在使用时，首先要检查是否有其他进程在临界区，确认没有时再进入。在程序中，P(mutex)和 V(mutex)用于实现对临界区的互斥，P(mutex)和 V(mutex)必须成对出现。

(2) 信号量 full 表示有数据的缓冲区的数目，初始值为 0。empty 表示空闲的缓冲区的数目，初值为 $n$。它们表示的都是资源的数目，因此称为资源信号量。实际上，full 和 empty 之间存在关系 full+empty=$n$。对资源信号量的 PV 操作同样需要成对出现，与互斥信号量不同的是，P 操作和 V 操作分别处于不同的程序中，例如 P(empty)在生产者进程中，而 V(empty)在消费者进程中。当生产者进程因执行 P(empty)而阻塞时，由消费者进程用 V(empty)将其唤醒；同理，当消费者进程因执行 P(full)而阻塞时，由生产者进程用V(full)将其唤醒。

(3) 多个 P 操作的次序不能颠倒。在程序中，应先对资源信号量执行 P 操作，再对互斥信号量执行 P 操作，否则可能引起死锁。

**4. 要思考的问题**

针对生产者—消费者问题，请读者从以下几个方面讨论各个进程的运行情况。

(1) 多个生产者进程运行，消费者进程未被调度运行。

(2) 多个消费者进程运行，生产者进程未被调度运行。

(3) 生产者和消费者进程交替被调度运行。

## 3.2.2 读者—写者问题

**1. 问题描述**

一个数据对象若被多个并发进程所共享，且其中一些进程只要求读该数据对象的内容，而另一些进程则要求写操作，对此，把只想读的进程称为“读者”，而把要求写的进程称为“写者”。在读者—写者问题中，任何时刻要求“写者”最多只允许有一个，而“读者”则允许有多个。因为多个“读者”的行为互不干扰，它们只是读数据，而不会改变数据对象的内容。而“写者”则不同，它们要改变数据对象的内容，如果它们同时操作，则数据对象的内容将会变得不可知。所以，对共享资源的读写操作的限制条件如下所述。

(1) 允许任意多读进程同时读。

(2) 一次只允许一个写进程进行写操作。

(3) 如果有一个写进程正在进行写操作，禁止任何读进程进行读操作。

**2. 用信号量解决读者—写者问题**

为了解决该问题，只需解决“写者与写者”和“写者与第一个读者”的互斥问题即可，为此引入一个互斥信号量 Wmutex。为了记录谁是第一个读者，可以用一个全局整型变量 Rcount 做一个计数器。而在解决问题的过程中，由于使用了全局变量 Rcount，该变量又是一个临界资源，对于它的访问仍需要互斥进行，所以需要一个互斥信号量 Rmutex。算法如下。

```
semaphore  Wmutex,Rmutex = 1;
int  Rcount = 0;

void  reader()              /* 读者进程 */
```

```
{
while  (true)
  {
    P(Rmutex);
    if  (Rcount == 0)  P(wmutex);
    Rcount = Rcount + 1;
    V(Rmutex);
    …;
    read;                    /* 执行读操作 */
    …;
    P(Rmutex);
    Rcount = Rcount - 1;
    if (Rcount == 0) V(wmutex);
    V(Rmutex);
  }
}


void  writer()               /*写者进程*/
{
while  (true)
  {
    P(Wmutex);
    …;
    write;                   /* 执行写操作 */
    …;
    V(Wmutex);
  }
}
```

**3. 要思考的问题**

对于读者—写者问题,有以下 3 种优先策略。

(1) 读者优先。即当读者进行读时,后续的写者必须等待,直到所有读者均离开后,写者才可进入。

前面的程序隐含使用了该策略。

(2) 写者优先。即当一个写者到来时,只有那些已经获得授权允许读的进程才被允许完成它们的操作,写者之后到来的新读者将被推迟,直到写者完成。在该策略中,如果有一个不可中断的连续的写者,读者进程会被无限期地推迟。

请读者思考如何修改前面的算法。

(3) 公平策略。以上两种策略,读者或写者进程中一个对另一个有绝对的优先权,Hoare 提出了一种更公平的策略,由如下规则定义。

① 规则 1:在一个读序列中,如果有写者在等待,那么就不允许新来的读者开始执行。

② 规则 2:在一个写操作结束时,所有等待的读者应该比下一个写者有更高的优先权。

对于该公平策略,又如何予以解决呢?

### 3.2.3 哲学家进餐问题

**1. 问题描述**

哲学家进餐问题是一个典型的同步问题，它由 Dijkstra 提出并解决。有 5 个哲学家，他们的生活方式是交替思考和进餐。哲学家们共用一张圆桌，围绕圆桌而坐，在圆桌上有 5 个碗和 5 支筷子，平时哲学家进行思考，饥饿时拿起其左、右两支筷子，试图进餐，进餐完毕又进行思考，如图 3-12 所示。这里的问题是哲学家只有拿到靠近他的两支筷子才能进餐，而拿到两支筷子的条件是他的左、右邻居此时都没有进餐。

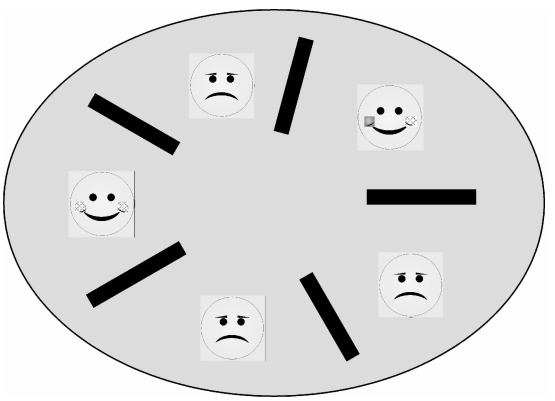

图 3-12 哲学家进餐问题

**2. 用信号量解决哲学家进餐问题**

由分析可知，筷子是临界资源，一次只允许一个哲学家使用。因此可以用互斥信号量来实现，描述如下。

```
semaphore  chopstick[5] = {1,1,1,1,1};
void  philosopher (int i)              /* 哲学家进程 */
{
while  (true)
  {
    P(chopstick[i]);
    P(chopstick[(i + 1) % 5]);
    …;
    eat;                               /* 进餐 */
    …;
    V(chopstick[i]);
    V(chopstick[(i + 1) % 5]);
    …;
    think;                             /* 思考 */
    …;
    }
}
```

**3. 算法潜在的问题**

在以上描述中,虽然解决了两个相邻的哲学家不会同时进餐的问题,但是有一个严重的问题,如果所有哲学家总是先拿左边的筷子,再拿右边的筷子,那么就有可能出现这样的情况,就是5个哲学家都拿起了左边的筷子,当他们想拿右边的筷子时,却因为筷子已被别的哲学家拿去,而无法拿到。此时所有哲学家都不能进餐,这就出现了死锁现象。

信号量在解决进程互斥和同步问题时是一个非常有效的工具,但是如果使用不当,可能引起死锁。

请读者思考,写出一个用信号量解决哲学家进餐问题且不产生死锁的算法。

## 3.2.4 打瞌睡的理发师问题

**1. 问题描述**

理发店有一名理发师,一把理发椅,还有 $N$ 把供等候理发的顾客坐的普通椅子。如果没有顾客到来,理发师就坐在理发椅上打瞌睡;当顾客到来时,就唤醒理发师;如果顾客到来时理发师正在理发,顾客就坐下来等待;如果 $N$ 把椅子都坐满了,顾客就离开该理发店去别处理发,如图3-13所示。要求为理发师和顾客各编写一段程序,描述他们的行为,并用信号量保证上述过程的实现。

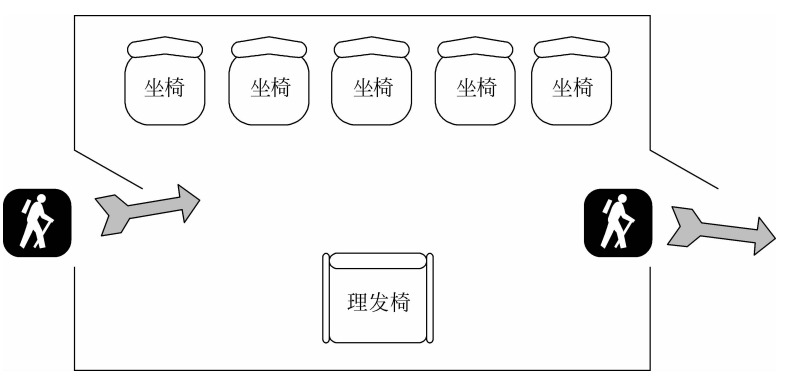


图3-13 打瞌睡的理发师问题

**2. 用信号量解决打瞌睡的理发师问题**

为理发师和顾客分别写一段程序,并创建进程。理发师开始工作时,先看看店里有无顾客,如果没有,则在理发椅上打瞌睡;如果有顾客,则为等待时间最长的顾客理发,且等待人数减1。顾客来到店里,先看看有无空位,如果没有空位,就不等了,离开理发店;如有空位,则等待,等待人数加1;如果理发师在打瞌睡,则将其唤醒。

为了解决上述问题,设一个计数变量 waiting,表示等候理发的顾客人数,初值为0;设3个信号量,customers 用来记录等候理发的顾客数(不包括正在理发的顾客);barners 用来记录正在等候顾客的理发师数(其值为0或1);mutex 用于互斥。程序描述如下。

```
#define CHAIRS 5                    /* 为等候的顾客准备的坐椅数 */
semaphore customers = 0;
semaphore barners = 0;
semaphore mutex = 1;
```

```
int waiting;
void barber()                    /* 理发师进程 */
{
  while (true)
  {
    P(customers);                /* 如果没有顾客,理发师就打磕睡 */
    P(mutex);                    /* 互斥进入临界区 */
    waiting-- ;
    V(barners);                  /* 理发师准备理发了 */
    V(mutex);
    cut_hair();                  /* 理发 */
  }
}

void customer ()                 /* 顾客进程 */
{
  P(mutex);
  if (waiting<CHAIRS)            /* 如果有空位,则顾客等待 */
  {
      waiting++;
      V(customers);              /* 如果有必要,唤醒理发师 */
      V(mutex);
      P(barners);                /* 如果理发师正在理发,则顾客等待 */
      get_haircut();
  }
  else                           /* 如果没有空位,则顾客离开 */
    V(mutex);
}
```

当有一个顾客来到理发店时,执行 customer()过程,首先获取信号量 mutex 进入临界区,如果不久另一个顾客到来,新到顾客只能等到释放 mutex 后才能进入。

进入临界区的顾客随后查看是否有椅子可坐,若没有,则释放 mutex 并离开; 若有椅子可坐,则对计数变量加 1,之后执行 V(customers)操作唤醒理发师。当顾客释放 mutex 后,理发师获得 mutex,他进行一些准备后开始理发。理发完毕,顾客退出 customer ()过程,离开理发店。

**3. 思考问题**

(1) 为什么理发师进程中使用循环语句,而顾客进程却没有?

(2) 程序中 waiting 的计数作用能否用信号量 customers 代替?

## 3.3 AND 信号量

3.2 节用信号量解决了很多同步和互斥问题,但在解决问题的过程中还存在一些其他问题,如在生产者和消费者问题中两个 P 操作的位置不能颠倒及哲学家进餐问题中的死锁

现象等。这些问题的出现促使AND信号量的产生,继而又发展到一般“信号量集”。

### 3.3.1 AND信号量的引入



**1. 问题的引出**

当利用信号量解决单个资源的互斥访问后,下面讨论控制进程对多个资源的互斥访问问题。在有些应用中,一个进程需要先获得两个或更多共享资源后方能执行其任务。假如有两个进程 $P_1$ 和 $P_2$,它们要共享两个全局变量 $R_1$ 和 $R_2$,为此要设置两个互斥信号量 $mutex_1$ 和 $mutex_2$,并令它们的初值为1;相应地,两个进程都要包含对信号量 $mutex_1$ 和 $mutex_2$ 的操作,假如操作如下。

```
/* 进程 P1 */                    /* 进程 P2 */
P(mutex1);                       P(mutex2);
P(mutex2);                       P(mutex1);
…                                …
```

如果进程 $P_1$ 和 $P_2$ 交替地执行P操作,则具体情况如下。

① 进程 $P_1$ 执行 $P(mutex_1)$,于是 $mutex_1=0$。

② 进程 $P_2$ 执行 $P(mutex_2)$,于是 $mutex_2=0$。

③ 进程 $P_1$ 执行 $P(mutex_2)$,于是 $mutex_2=-1$,进程 $P_1$ 阻塞。

④ 进程 $P_2$ 执行 $P(mutex_1)$,于是 $mutex_1=-1$,进程 $P_2$ 阻塞。

此时,两个进程处于僵持状态,都无法继续运行。

**2. AND信号量**

AND信号量同步机制就是要解决上述问题,其基本思想是将进程在整个运行期间所需要的所有临界资源一次性全部分配给进程,待该进程使用完后再一起释放。只要尚有一个资源不能满足进程的要求,其他所有能分配给该进程的资源也都不予以分配,为此在P操作上增加一个AND条件,故称为AND信号量。P操作的原语为Swait,V操作的原语为Ssignal。在Swait中,各个信号量的次序并不重要,尽管会影响进程进入哪个等待队列。由于Swait实施对资源的全部分配,进程获得全部资源并执行之后再释放全部资源,因此避免了前文所述的僵持状态。如下是Swait和Ssignal的伪代码。

```
Swait(s1,s2,…,sn)
{
    if (s1>=1 && s2>=1 && … && sn.>=1)
    { /* 满足资源要求时 */
      for (i=1;i<=n;i=i+1)
          si=si-1;
    }
    else
    { /* 某些资源不能满足要求时 */
       将进程投入第一个小于1的信号量的等待队列 si.queue;
       阻塞进程;
    }
}
```

```
Ssignal(s1,s2,…,sn)
{  for (i=1;i<=n;i=i+1)
    {
    si=si+1;
    for (等待队列 si.queue 中的每个进程 P)
      {
        if (进程 P 通过 Swait 中的测试)
            {/* 通过检查,即资源够用 */
                唤醒进程 P,将 P 投入就绪队列;
            }
            else
            {/* 未通过检查,即资源不够用 */
            进程 P 进入某等待队列;
            }
      }
    }
  }
}
```

### 3.3.2 用 AND 信号量解决实际应用

**1. 用 AND 信号量解决哲学家进餐问题**

下面讨论用 AND 信号量解决哲学家进餐问题。在该问题中,筷子是临界资源,而题目中的临界资源有 5 个,每个哲学家需要拿到两个临界资源才可以进餐,所以为了避免死锁的产生,哲学家在申请临界资源时必须一次性申请其所需要的所有资源。具体解法如下。

```
semaphore  chopstick[5]={1,1,1,1,1};
void  philosopher()          /*哲学家进程*/
{
  while  (true)
  {
    Swait(chopstick[i],chopstick[(i+1) % 5]);
    …;
    eat;                     /* 进餐 */
    …;
    Ssignal(chopstick[i],chopstick[(i+1) % 5]);
    …;
    think;                   /* 思考 */
    …;
    }
}
```

**2. 用 AND 信号量解决生产者—消费者问题**

用 AND 信号量解决生产者—消费者问题。对于生产者—消费者问题,用 AND 信号量来解决,可以避免因 P 操作的次序错误而发生死锁现象,程序描述如下。

```
semaphore  mutex = 1;
semaphore  empty = n;
semaphore  full = 0;
int  i,j;
ITEM  buffer[n];
ITEM  data_p,data_c;

void  producer()              /* 生产者进程 */
{
while  (true)
    {
      produce an item in data_p;
      Swait(empty,mutex);
      buffer[i] = data_p;
      i = (i + 1)  %  n;
      Ssignal(mutex,full);
    }
}

void  consumer()              /* 消费者进程 */
{
while  (true)
    {
      Swait(full,mutex);
      data_c = buffer[j];
      j = (j + 1)  %  n;
      Ssignal(mutex,empty);
      consume the item in data_c;
    }
}
```

程序中用 Swait(empty,mutex)代替了 P(empty)和 P(mutex),用 Ssignal(mutex,full)代替了 V(mutex)和 V(full),用 Swait(full,mutex)代替了 P(full)和 P(mutex),用 Ssignal(mutex,empty)代替了 V(mutex)和 V(empty);对信号量的操作同时进行,避免了死锁。

## 3.4 管　　程

用信号量可以实现进程之间的同步和互斥,但要设置很多信号量,使用大量 P、V 操作,还要仔细安排多个 P 操作的排列次序,否则将出现错误的结果或死锁现象。为了解决这些问题,可以使用另一高级同步工具——管程。

### 3.4.1 管程的思想

Dijkstra 于 1971 年提出,把所有进程对某一临界资源的同步操作集中起来,构成一个所谓的"秘书"进程。凡是要访问临界资源的进程,都必须先向"秘书"报告,并由"秘书"实现诸进程的同步。1973 年,Hansan 和 Hoare 又把"秘书"的思想发展为管程的概念,把并发进

程之间的同步操作分别集中于相应的管程中。管程思想在许多程序设计语言中得到了实现，包括并发 Pascal、Pascal_Plus、Modula_2、Modula_3 和 Java。

**1. 管程的概念**

管程的定义是一个共享资源的数据结构以及一组能为并发进程在其上执行的针对该资源的一组操作，这组操作能同步进程和改变管程中的数据。

管程的基本思想是把信号量及其操作原语封装在一个对象内部，即将共享资源以及针对共享资源的所有操作集中在一个模块中。管程可以用函数库的形式实现，一个管程就是一个基本程序单位，可以单独编译。

**2. 管程的特征**

管程的主要特征有以下几点。

(1) 局限于管程的共享变量(数据结构)只能被管程的过程访问，任何外部过程都不能访问。

(2) 一个进程通过调用管程的一个过程进入管程。

(3) 任何时候只能有一个进程在管程中执行，调用管程的任何其他进程都被挂起，以等待管程变为可用，即管程有效地实现互斥。

上述前两个特征就像面向对象软件中的对象，即管程中引入了面向对象的思想，一个管程不仅有关于共享资源的数据结构，而且还有对数据结构进行操作的代码。

管程对共享资源进行了封装，进程可以调用管程中定义的操作过程，而这些操作过程的实现在管程的外部是不可见的。管程相当于围墙，它把共享资源和对它的操作的若干过程围了起来，所有进程要访问临界资源时，都必须经过管程(相当于通过围墙的门)才能进入，而管程每次只允许一个进程进入，从而实现了进程互斥。进入管程的互斥机制是由编译器负责的，通常使用信号量。由于实现互斥是由编译器完成的，不用程序员自己实现，所以出错的概率很小。

## 3.4.2 管程的结构

为了实现并发，管程必须包含同步工具。假设一个进程调用了管程，当它在管程中时要等待某个条件，条件不满足时它必须被挂起。这就需要一种机制，使得该进程不仅能被挂起，而且当条件满足且管程再次可用时，可以恢复该进程，并允许它在挂起点重新进入管程。

**1. 条件变量**

管程必须使用条件变量提供对同步的支持，这些条件变量包含在管程中，并且只有在管程中才能被访问。以下两个函数可以操作条件变量。

(1) cwait(c)：调用进程的执行在条件 c 上挂起，管程现在可被另一个进程使用。

(2) csignal(c)：恢复在 cwait 上因为某些条件而挂起的进程的执行。如果有多种这样的进程，选择其中一个。

注意，管程中的条件变量不是计数器，不能像信号量那样积累信号，供以后使用。如果在管程中的一个进程执行 csignal(c)，而在条件变量 c 上没有等待着的进程，则它所发送的信号将丢失。换句话说，cwait(c)操作必须在 csignal(c)操作之前，这条规则使实现更加简单。

**2. 管程的结构**

图 3-14 给出了管程的结构。尽管一个进程可以通过调用管程中的任何一个过程进入管程,仍可以把管程想象成具有一个入口点,并保证一次只有一个进程可以进入。其他试图进入管程的进程加入挂起等待管程可用的进程队列。但一个进程在管程中时,它可能会通过发送 cwait(x)把自己暂时挂起在条件 x 上,随后它被放入等待条件改变以重新进入管程的进程队列中。

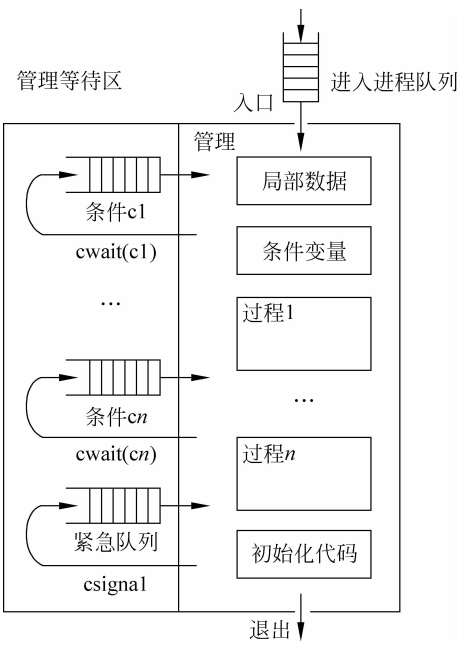


图 3-14 管程的结构

如果在管程中执行的一个进程发现条件变量 x 发生了变化,它就发送 csignal(x),通知相应的条件队列条件已经改变。

## 3.4.3 用管程解决实际应用

**1. 生产者—消费者问题**

管程模块 monitor_PC 控制着用于保存和取回字符的缓冲区,管程中有两个条件变量 notfull 和 notempty。当缓冲区中只要有一个字符的空间时,notfull 为真;当缓冲区中至少有一个字符时,notempty 为真。

```
/* 用管程解决生产者和消费者问题 */
monitor  monitor_PC;
char  buffer[n];                    /* 缓冲区的大小为 n */
int  nextin,nextout;                /* 指向缓冲区的指针 */
int  count;                         /* 缓冲区计数器 */
condition  notfull,notempty;        /* 用于同步的条件变量 */

void  put(char x);                  /* 存数据过程 */
```

```
{
  if (count == n) cwait(notfull);          /* 缓冲区满,防止溢出 */
  buffer[nextin] = x;
  nextin = (nextin + 1) % n;
  count = count + 1;
  csignal(notempty);                       /* 恢复一个正在等待的消费者 */
}
void  get(char x);                         /* 取数据过程 */
{
  if (count == 0) cwait(notempty);         /* 缓冲区空,防止下溢 */
  x = buffer[nextout];
  nextout = (nextout + 1) % n;
  count = count - 1;
  csignal(notfull);                        /* 恢复一个正在等待的生产者 */
}
{                                          /* 管程体 */
  nextin = 0; nextout = 0; count = 0;      /* 变量初始化 */
}

void producer()                            /* 生产者进程 */
{
  char x;
  while (true)
  {
     produce an char in x;
     monitor_PC.put(x);
  }
}
void consumer()                            /* 消费者进程 */
{
  char  x;
  while (true)
{
   monitor_PC.get(x);
   consume an x;
   }
}
```

生产者可以通过管程中的过程 put()往缓冲区中增加字符,它不能直接访问 buffer。put()过程首先检查条件 notfull,以确定缓冲区是否还有可用空间。如果没有,执行管程的进程在这个条件上被挂起。其他某个进程(消费者)现在可以进入管程。当缓冲区不再满时,被挂起的进程从队列中移出,被激活后重新执行。当往缓冲区放置一个字符后,该进程发送 notempty 条件信号,对消费者的处理类似。

**2. 管程与信号量的区别**

从上文实例可以看出,与信号量相比,管程担负的责任不同,管程构造了自己的互斥机

制,就是生产者和消费者不可能同时访问缓冲区。但是要求程序员必须把 cwait 和 csignal 原语放到管程中合适的位置,用于防止进程往一个满的缓冲区中存放数据,或从一个空缓冲区中取数据,而在使用信号量时,互斥和同步都属于进程的责任。

在管程的程序中,进程执行 csignal 后立即退出管程,如果在过程最后没有发生 csignal,Hoare 建议把发送该信号的进程挂起,从而使管程可用,并把挂起进程放入队列,直到管程空闲。此时,一种可能是把挂起进程放置到入口队列中,这样它必须与其他没有进入管程的进程竞争。但是,由于在 csignal 上挂起的进程已经在管程中执行了部分任务,因此使它们优先于新进入的进程是很有意义的,这可以通过建立一条独立的紧急队列来实现。并发 Pascal 是使用管程的一种计算机语言,它要求 csignal 只能作为管程过程中执行的最后一个操作。

如果没有进程在条件 x 上等待,csignal(x)的执行将不会产生任何效果。

而对于信号量,如果在同步操作中省掉任何一个信号操作,那么进入相应条件队列的进程将会永远被挂起。管程优于信号量之处在于所有同步机制都被限制在管程内部,因此易于验证同步的正确性,易于检查出错误。此外,如果有一个管程被正确地使用,则所有进程对受保护资源的访问都是正确的;而对于信号量,只有当所有访问资源的进程都能正确地使用信号量时,资源访问才能保证正确。

## 3.5 同步与互斥实例

前文讨论的进程同步与互斥问题在具体的某个操作系统中又是如何使用的呢?下面讨论 Solaris、Windows、Linux 等操作系统所使用的同步机制。

### 3.5.1 Solaris 的同步与互斥

为了控制对临界区的访问,Solaris 提供了自旋锁、信号量、管程、读写锁和十字转门几种方法。信号量和管程已在前文介绍,这里不再赘述。本节介绍自旋锁、读写锁和十字转门。

**1. 自旋锁**

保护临界区最常见的技术是自旋锁。在同一时刻,只有一个线程能获得自旋锁。其他企图获得自旋锁的任何线程将一直进行尝试(即自旋),直到获得该锁。本质上,自旋锁建立在内存区中的一个整数上,任何线程进入临界区之前都必须检查该整数。如果该值为 0,则线程设置该值为 1,然后进入临界区。如果该值非 0,则该线程继续检查该值,直到它为 0。

在单处理机系统中,如果线程碰到锁,将总是进入阻塞状态,而不是自旋,因为单处理机上任何时刻只有一个线程在运行。Solaris 使用自旋锁方法保护那些只有几百条指令的短代码段的临界数据。因为如果代码段较长,自旋等待将很低效。所以长代码段使用管程和信号量比较好。

**2. 读写锁**

读写锁允许在内核中实现比自旋锁更高的并发度。读写锁允许多个线程同时以只读的方式访问同一数据结构,只有当一个线程想要更新数据结构时,才会互斥地访问该自旋锁。

读写锁用于保护经常访问但通常是只读访问的数据。在这种情况下,读写锁比信号量更有效,因为多个线程可以同时读数据,而信号量只允许顺序访问数据。

**3. 十字转门**

十字转门是一个等待队列，队列中的线程是阻塞在锁上的线程。Solaris 使用十字转门管理等待在适应互斥和读写锁上的线程链表。例如，如果一个线程拥有锁，那么其他试图获取锁的线程就会阻塞并进入十字转门。当锁被释放时，内核会从十字转门中选择一个线程作为锁的下一个拥有者。Solaris 管理十字转门的不同点是系统不是将每个互斥对象与一个十字转门相关联，而是给每个内核线程一个十字转门。这是因为一个线程某一时刻只能阻塞在一个对象上，所以这比每个对象都有一个十字转门更有效。

第一个阻塞于某个互斥对象的线程的十字转门成为对象的十字转门，以后所有阻塞于该锁上的线程将增加到该十字转门中。当最初的线程被释放时，它会从内核所维护的空闲十字转门中获得一个新的十字转门。

## 3.5.2 Windows 的同步与互斥

Windows 系统采用多线程机制，并支持实时应用程序和多处理机。在单处理机上，当线程访问某个全局资源时，它暂时屏蔽所有可能访问该全局资源的中断。在多处理机上，Windows 采用自旋锁来保护对全局资源的访问。与 Solaris 一样，内核使用自旋锁来保护较短的代码段，并且内核保证拥有自旋锁的线程不会被抢占。

**1. 屏蔽中断**

在单处理系统中，最简单的方法是使每个进程在刚刚进入临界区后立即屏蔽所有中断，并在就要离开之前再打开中断。屏蔽中断后，时钟中断也被屏蔽。CPU 只有发生时钟中断或其他中断时才会进行进程切换，这样在屏蔽中断之后 CPU 将不会被切换到其他进程。于是，一旦某个进程屏蔽中断，它就可以检查和修改共享内存，而不必担心其他进程介入。

**2. 调度对象**

对于内核外线程的同步和互斥，Windows 系统提供了调度对象。采用调度对象，线程可根据多种不同机制，包括互斥、信号量、事件和定时器等，来实现同步和互斥。互斥和信号量已在前文介绍，这里不再赘述。事件是一个同步机制，其使用与管程中的条件变量相似，即当条件出现时会通知等待线程。定时器用来在一定事件后通知一个或多个线程。

调度对象可以处于触发状态或非触发状态。触发状态表示对象可用，且线程获取它时不会阻塞。非触发状态表示对象不可用，且当线程试图获取它时会阻塞。

调度对象的状态和线程状态有一定的关系。当线程阻塞在非触发调度对象时，其状态从就绪转变为阻塞，且该线程被放到对象的等待队列上。当调度对象为触发时，内核检查有没有线程在该对象上等待，如果有，则内核将改变一个或多个线程的状态，使其从阻塞状态切换为就绪状态，以重新获得运行的机会。内核从等待队列中选择的线程的数量与它们所等待对象的调度类型有关。对于互斥，内核只从等待队列中选择一个线程，因为一个互斥对象只能为单个线程拥有。对于事件对象，内核可以选择多个所有等待事件的线程。

## 3.5.3 Linux 的同步与互斥

Linux 2.6 以前的版本为非抢占式内核，即纵然有更高优先级的进程也不能抢占正在运行的其他进程。然而现在的 Linux 为抢占式，即使在内核态下运行的进程也可以被抢占。

Linux 内核的同步和互斥机制除了常规的管道、消息、共享内存和信号之外，还采用了

屏蔽中断、自旋锁、读写锁和信号量。

对于单处理机不适合使用自旋锁,因此采用禁止和允许内核抢占来实现。对于对称多处理机,则采用自旋锁。Linux 提供了两个系统调用 preempt_disable 和 preempt_enable 来禁止和允许内核抢占。

自旋锁和禁止与允许内核抢占适用于短代码段,对于长时间使用的临界数据,使用信号量更合适。

## 3.6 进 程 通 信

进程间的通信要解决的问题是进程之间信息的交流,这种信息交流的量可大可小。操作系统提供了多种进程通信的机制,可分别适用于多种不同的场合。前面介绍的进程同步就是进程通信的一种形式,只不过交流的信息量非常少。按交换信息量的大小,可以把进程之间的通信分成低级通信和高级通信。

在低级通信中,进程之间只能传递状态和整数值,信号量机制属于低级通信方式,低级通信方式的优点是传递信息的速度快,缺点是传送的信息量少、通信效率低。如果要传递较多的信息,就需要多次通信完成,用户直接实现通信的细节编程复杂,容易出错。

在高级通信中,进程之间可以传送任意数量的数据,传递的信息量大,操作系统隐藏了进程通信的实现细节,大大简化了进程通信程序编制上的复杂性。

### 3.6.1 进程通信的类型

随着操作系统的发展,进程之间的通信机制也得到很大发展,高级通信机制可分为三大类,分别为共享存储器系统、消息传递系统和管道通信。

**1. 共享存储器系统**

在共享存储器系统中,相互通信的进程共享某些数据结构或存储区域,进程之间通过共享的存储区域进行通信。

进程通信前,向系统申请共享存储区域,并指定该共享区域的名称,若系统已经把该共享区域分配给其他进程,则将该共享区域的句柄返回给申请者。申请进程把获得的共享区域连接在本进程上之后,便可以像读写普通存储区域一样对该共享存储区域进行读写操作,以达到传递大量信息的目的。

**2. 消息传递系统**

在消息传递系统中,进程间的数据交换以消息为单位。用户通过使用操作系统提供的一组消息通信原语来实现信息的传递。消息传递系统是一种高级通信方式,它因实现方式不同又可以分为直接通信方式和间接通信方式。

1) 直接通信方式

该方式下,发送方直接将消息发送给接收方,接收方可以接收来自任意发送方的消息,并在读出消息的同时得知发送者是谁。

2) 间接通信方式

在这种方式下,消息不是直接从发送方发送到接收方,而是发送到临时保存这些消息的队列,这个队列通常称为信箱。因此,两个通信进程中,一个给一个合适的信箱发消息,另一

个从信箱中获得这些消息。

间接通信方式在消息的使用上有很大的灵活性。发送方和接受方之间的关系可以是一对一、多对一、一对多和多对多。

(1) 一对一的关系可以在两个进程之间建立专用的通信链接,这可以把它们之间的交互隔离起来,避免其他进程的干扰。

(2) 多对一的关系对客户端/服务器之间的交互非常有用,系统中有多个客户端进程和一个服务器进程。服务器进程给多个客户端进程提供服务,这时,信箱常常称做是一个端口。

(3) 一对多的关系适用于一个发送方和多个接收方,它对于在一组进程之间广播一条消息或某些信息的应用非常有用。

(4) 多对多的关系一般用在共用信箱中,让多个进程都能向信箱中投递消息,也可从信箱中取走自己的消息。

进程和信箱的关联可以是静态的,也可以是动态的。端口常常是静态地关联到一个特定的进程上,也就是说,端口是永久被创建并指定到该进程。当有许多发送者时,发送者和信箱间的关联可以是动态发生的,基于这个目的,可以使用如 connect 和 disconnect 这样的原语进行显式的连接。

另一个问题是信箱的所有权问题。对于端口,它通常归接收进程所有,并由接收进程创建。因此,当接收进程被撤销时,它的端口也随之被撤销。对于通用的信箱,操作系统可以提供一个创建信箱的服务,这样信箱可以看做由创建它的进程所有,在这种情况下它们也同该进程一起终止;或者把信箱看做由操作系统所有,此时要撤销信箱需要一个显式的命令。

**3. 管道通信**

所谓管道,是指用于连接一个读进程和一个写进程,以实现进程之间通信的一种共享文件,又称为 Pipe 文件。向管道提供输入的是发送进程,或称为写进程,它负责向管道送入数据,数据的格式是字符流;而接收管道数据的接收进程称为读进程。由于发送进程和接收进程是利用管道来实现通信的,所以称为管道通信。管道通信始创于 UNIX 系统,因它能传送大量数据,且很有效,故目前许多操作系统(如 Windows 2000、Linux、OS/2)都提供管道通信。

为了协调双方的通信,管道通信机制必须提供以下几个方面的协调能力。

(1) 互斥。当一个进程正在对管道进行读或写操作时,另一个进程必须等待。

(2) 同步。管道的大小是有限的。所以当管道满时,写进程必须等待,直到读进程把它唤醒为止。同理,当管道没有数据时,读进程也必须等待,直到写进程将数据写入管道后,读进程才被唤醒。

(3) 对方是否存在。只有确认对方存在时,方能进行通信。

## 3.6.2 进程通信中的问题

进程通信中需要考虑的问题有通信链路、数据格式和进程的同步方式等。

**1. 通信链路的建立方式**

为了使发送进程和接收进程之间能够进行通信,必须在它们之间建立一条通信链路。建立通信链路的方式有两种,即显式建立链路和隐式建立链路。显式建立链路就是在发送

进程发送信息之前用一个“建立连接”的显式命令建立通信链路,当链路使用完毕后再利用显式命令的方式将链路拆除。隐式建立链路是在进程进行通信时发送进程不必明确地提出建立链路的请求,直接利用系统提供的发送原语进行信息的传递,此时操作系统会自动地为之建立一条链路,无须用户操心。一般地说,网络通信常常用显式的方式建立链路,本机进程通信采用隐式的方式建立通信链路。

**2. 通信方向**

根据通信的方向,进程通信又可以分为单向通信方式和双向通信方式。单向通信方式是指只允许发送进程向接收进程发送消息,反之不行。双向通信方式允许一个进程向另外一个进程发送消息,也可以反过来由另一个进程向发过消息的进程回送消息。双向通信方式由于进程之间可以对发过的消息进行回送确认,因此比较可靠。

**3. 通信链路连接方式**

根据通信链路的连接方式可以把通信链路分为点对点连接方式和广播方式。其中,点对点方式指用一条链路将两个进程进行链接,通信的完成只与这两个进程有关。广播方式是指一条链路上连接了多个(大于两个)进程,其中一个进程向其他多个进程同时发送消息。

**4. 通信链路的容量**

链路的容量是指通信链路上是否有用于暂存数据的缓冲区。无容量通信链路上没有缓冲区,因而不能暂存任何消息。而有容量通信链路是指在链路中设置了缓冲区,因而可以暂存消息,缓冲区的数目越大,通信链路的容量越大。

**5. 数据格式**

数据格式主要分成字节流和报文两种。采用字节流方式时,发送方发送的数据没有一定的格式,接收方不需要保留各次发送之间的分界。而报文方式就比较复杂了,通常把报文分为报头和正文两部分。报头包括数据传输时所需的控制信息,如发送进程名、报文的长度、数据类型、数据的发送日期和时间等。而正文部分才是真正要发送的信息。另外在报文方式中,根据报文的长度又进一步分成定长报文和不定长报文。

**6. 同步方式**

根据收发进程在进行收发操作时是否等待,同步方式又分成两种,即阻塞方式和非阻塞方式。阻塞方式是指操作方要等待操作的结束。非阻塞方式指操作方在提交后立即返回,不需要等待。具体地说,一个进程向另一个进程发送消息后,发送进程可能有两种选择,一是自己阻塞,并等到接收方接收到消息后才被唤醒;另一种选择是继续执行。对于接收进程也类似。

### 3.6.3 消息传递系统的实现

消息传递系统首先由 Hansan 提出,并在 RC4000 系统上实现。在这种机制中,发送消息利用发送原语 send 实现,接收消息用原语 receive 实现。

**1. 消息传递系统的数据结构**

在消息传递系统中,主要使用的数据结构是消息缓冲区。其描述如下。

```
struct  message_buffer
{
  char  sender[30];                    /* 发送进程标识符 */
```

```
  int  size;                          /* 消息长度 */
  char  text[200];                    /* 消息正文 */
  struct  message_buffer *next;       /* 指向下一个消息缓冲区的指针 */
}
```

在使用消息传递系统时,需要使用信号量来保证消息缓冲区的互斥和协调发送进程与接收进程的同步,需要的数据结构如下。

```
struct  process_control
{
  struct  message_buffer *mq;         /* 消息队列队首指针 */
  semaphore  mutex;                   /* 消息队列互斥信号量,初值为 1 */
  semaphore  sm;                      /* 消息队列同步信号量,记录消息的个数 */
/* 初值为 0 */
}
```

**2. 发送原语**

发送进程在利用发送原语发送消息之前,应先在自己的内存空间设置一发送区 a(见图 3-15),把待发送消息的正文、长度及发送进程的标识符填入其中。然后调用发送原语,把消息发送给目标进程。在发送原语中,首先根据发送区 a 消息的长度申请一缓冲区 i;接着将发送区 a 的内容复制到消息缓冲区 i;为了能将消息 i 挂到接收进程的消息队列 mq 上,应先获得接收进程的进程标识符 j,然后将 i 挂到的消息队列 j.mq 上;由于该队列是临界资源,所以在 insert 操作的前后使用 P、V 操作和互斥信号量 mutex;最后使用 V(j.sm)唤醒接收进程,通知它可以接收消息了。

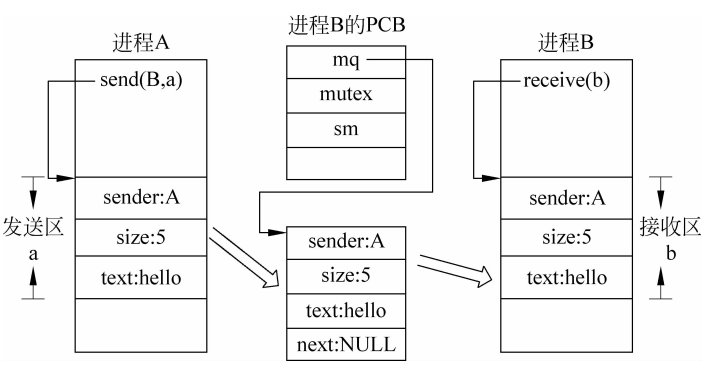


图 3-15　消息缓冲队列

```
void  send(receiver,a)
char  receiver[30];
struct  message_buffer a;
{
  struct message_buffer i;
  struct process_control j;
  getbuf(a.size,i);                   /* 根据发送区 a 消息的长度申请一缓冲区 i */
  i.sender = a.sender;
  i.size = a.size;
```

```
    i.text = a.text;
    i.next = NULL;
    getid(PCB_set,receiver,j);          /* 获得接收进程的进程标识符 j */
    P(j.mutex);
    Insert(j.mq,i);                     /* 将消息缓冲区 i 挂到的消息队列 j.mq 上 */
    V(j.mutex);
    V(j.sm);
  }
```

**3. 接收原语**

接收进程使用 receive()原语从自己的消息缓冲队列 mq 中取下一个消息 i,并将其中的数据复制到自己的消息接收区 b。

```
void  receive(b)
struct  message_buffer  b;
{
  struct  message_buffer  i;
  struct  process_control  j;
  j = internal_name();                  /* 接收进程的内部标识符 */
  P(j.sm);
  P(j.mutex);
  remove(j.mq,i);                       /* 从消息队列中摘下第一个消息缓冲区 */
  V(j.mutex);
  b.sender = i.sender;
  b.size = i.size;
  b.text = i.text;
}
```

### 3.6.4 客户端—服务器系统通信

用户想要访问的数据可能放在网络中的某个服务器上。例如,用户统计一个存放在服务器 A 上的文件的行数、字数。这个请求由远程服务器 A 来处理,它对文件进行统计,计算出所需要的结果,最后将结果数据送给用户。

在客户端—服务器系统中,常用的通信方式有命名管道、套接字和远程过程调用。

**1. 命名管道**

命名管道是客户端—服务器系统中一种可靠的双向通信机制,它由命名管道服务器和命名管道客户端组成。命名管道的创建由服务器一方负责,它只能在本机上建立命名管道;命名管道建立后,客户端可以连接到其他计算机的有名管道上;之后通信双方就可以像普通文件的读、写那样通过读、写管道来完成客户端—服务器两方进程的通信。

命名管道是无名管道在网络环境中的一种推广。

**2. 套接字**

套接字(Socket)既可用于同一台计算机上的两个进程之间的通信,也适用于网络环境下的进程通信,自 20 世纪 80 年代起成为 Internet 上的通信标准。

套接字由 IP 地址和端口号组成,IP 地址用于确定网络上的一台计算机,端口号用于确

定该计算机上的一个进程。

套接字采用客户端—服务器模式，服务器进程通过监听指定端口等待即将到来的客户端请求。一旦收到客户端请求，服务器就与客户端建立连接。当客户端进程发出连接请求时，它将得到一个端口号，该端口号保证所有连接都唯一地确定一个服务器进程和一个客户端进程；当客户端进程和服务器进程建立连接后，就可以交换无结构的字符流，字符流的解释与构造由客户端和服务器应用程序负责。

**3. 远程过程调用**

远程过程调用(Remote Procedure Call，RPC)是远程服务一种最常见的形式，它起源于20世纪80年代。

RPC采用客户端—服务器模式，其思想很简单，就是允许程序调用网络中其他计算机上的过程。请求程序就是一个客户端，而服务提供程序就是一个服务器。首先，调用进程发送一个有参数过程调用到服务进程，然后挂起自己，等待应答信息。在服务器端，进程保持睡眠状态，直到调用信息到达为止。当一个调用信息到达时，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息。最后，客户端调用过程接收答复信息，获得进程结果，然后调用进程继续执行。RPC把在网络环境中的过程调用所产生的各种复杂情况都隐藏起来，对于RPC应用程序，程序员无须编写任何代码来传输网络请求、选择网络协议、处理网络错误、等待结果等。RPC软件自动完成这些工作。

另外，在Windows系统中，如果通信的两个进程位于同一台机器上，就使用本地过程调用(Local Procedure Call，LPC)。LPC是一种消息传递工具，Windows使用端口对象建立和维护两个进程之间的连接。特别需要注意的是，LPC并不是Win32 API的一部分，所以也不能被应用程序员所见。应用程序员使用只能使用Win32 API调用标准的RPC，当RPC被同一机器上的进程所调用时，RPC通过本地过程调用被间接地处理。

**4. 远程方法调用**

远程方法调用(Remote Method Invocation，RMI)是一个类似RPC的Java特性。RMI允许线程调用远程对象的方法。如果对象位于不同的Java虚拟机上，那么就认为它是远程的。因此这里的远程可能是指在同一计算机上或通过网络连接的主机的不同Java虚拟机上。

RMI与RPC的在两个方面有所不同。第一，RPC支持子程序编程，即只能调用远程的子程序或函数；而RMI是基于对象的，它支持调用远程对象的方法。第二，在RPC中，远程过程的参数是普通的数据结构，而RMI可以将对象作为参数传递给远程方法。RMI通过允许Java的程序调用远程对象的方法，使得用户能够开发分布在网络上的Java应用程序。

## 思考与练习题

1. 以下进程之间存在相互制约关系吗？若存在，是什么制约关系？为什么？

(1) 几个同学去图书馆借同一本书。

(2) 篮球比赛中两队同学争抢篮板球。

(3) 果汁生产流水线中捣碎、消毒、灌装、装箱等各道工序。

(4) 商品的入库和出库。

(5) 工人做工与农民种粮。

2. 在操作系统中引入管程的目的是什么？条件变量的作用是什么？

3. 说明 P、V 操作为什么要设计成原语。

4. 设有一个售票大厅可容纳 200 人购票，如果厅内不足 200 人则允许进入，超过则在厅外等候；售票员某时只能给一个购票者服务，购票者买完票后就离开。

(1) 购票者之间是同步关系还是互斥关系？

(2) 用 P、V 操作描述购票者的工作过程。

5. 进程之间的关系如图 3-16 所示，试用 P、V 操作描述它们之间的同步。

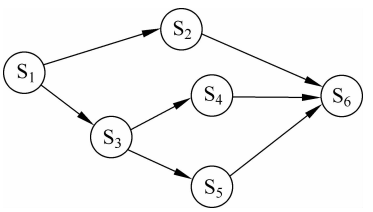


图 3-16　进程之间的关系

6. 有 4 个进程 $P_1$、$P_2$、$P_3$ 和 $P_4$ 共享一个缓冲区，进程 $P_1$ 向缓冲区中存入消息，进程 $P_2$、$P_3$ 和 $P_4$ 从缓冲区中取消息，要求发送者必须等 3 个进程都取过本条消息后才能发送下一条消息。缓冲区内每次只能容纳一个消息，用 P、V 操作描述 4 个进程存取消息的情况。

7. 分析生产者—消费者问题中多个 P 操作颠倒引起的后果。

8. 读者—写者问题中写者优先算法的实现。

9. 写一个用信号量解决哲学家进餐问题又不产生死锁的算法。

10. 一个文件可由若干个不同的进程所共享，每个进程具有唯一的编号。假定文件可由满足下列限制的若干个进程同时访问，并发访问该文件的那些进程的编号的总和不得大于 $n$，设计一个协调对该文件访问的管程。

11. 用管程解决读者—写者问题，并采用公平原则。