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。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，进程被交替地执行，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了同一个全局变量，并且都对该变量进行了读写操作，那么不同的读写执行顺序是非常关键的。关于这个问题会在本节的下面举例说明。

(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。这是因为并发程序的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)个问题已经在上一章解决,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,本节重点要解决的是第(4)个问题。

2. 进程的交互

这里以进程之间是否知道对方的存在,按进程的交互方式划分,可以分为以下三种情况。

(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 列出了三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 死锁 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 死锁 饿死 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 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。

从以上可以看出,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 三个进程,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,这三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

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,在当前的进程使用完毕之前,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) 两个进程交替地执行,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}。执行完毕,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。

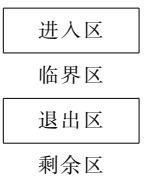

图 3-1 进程访问临界区的一般结构

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,但所有的同步机制都必须遵循下述四个准则。

(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。此时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。

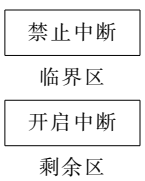

图 3-2 用禁止中断的方法访问临界区

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,进程被限制只能接替执行。

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 指令的功能可以用下面的函数来描述。

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

```
while (TS(lock))
  /* 什么也不做 */;
```

临界区;

```
lock=false;
```

剩余区;

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,可以用下面的函数描述 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

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。

但是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，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中提供某种支持。

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。

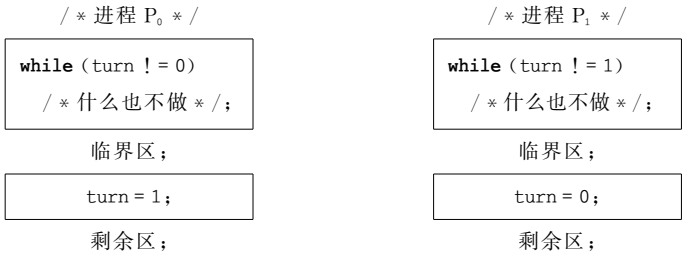

图 3-5　单标志算法

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，表示进程 P_0 进入临界区执行。

在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，否则才可进入，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，因而造成了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地“谦让”而都进不了临界区。

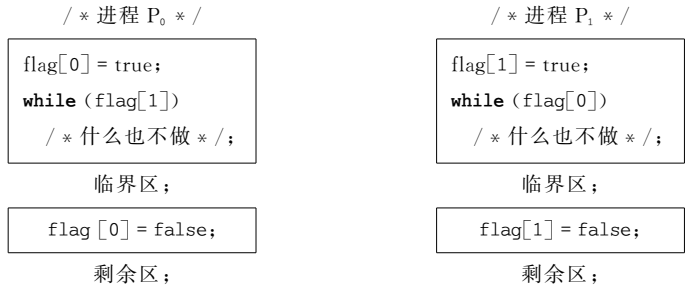

图 3-7　双标志、先修改后检查算法

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，如果对方不想进入时，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。

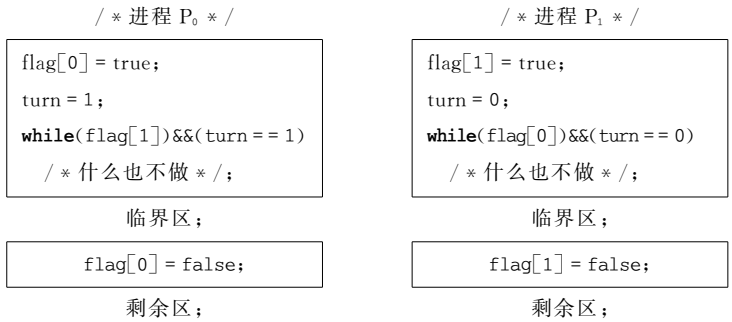

图 3-8　先修改、后检查、后修改者等待算法

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，对于三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，有一定的难度，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，信号量的定义为：

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

P 原语所执行的操作可用下面的函数 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,并插入到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。

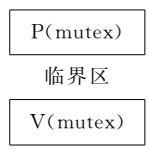

图 3-9　用信号量解决互斥问题

例如,对于前面 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，即：

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

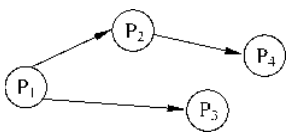

图 3-10　四个进程之间的执行次序

对于图 3-10 所示的进程关系：有四个并发执行的进程 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 才能执行；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，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)。一组进程被称为生产者，另一组进程被称为消费者。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。生产者进程不断地将生产的产品放入缓冲池，消费者进程不断地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)。

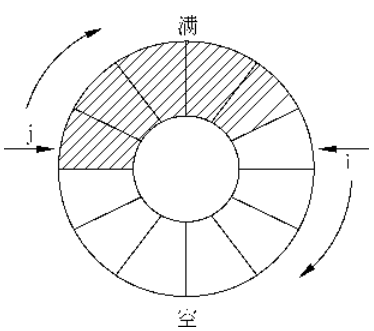

图 3-11　环形缓冲池

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，而进入等待状态，同时唤醒消费者；当所有的缓冲区都空时，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，所有的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 操作解决生产者和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. 注意的问题

在生产者—消费者问题中要注意以下几个问题。