第3章 进程并发控制

多道程序设计技术大大提高了系统的资源利用率，但是使多个程序能正确地并发执行是相当困难的。并发是所有问题的基础，也是操作系统设计的基础。并发包括很多设计问题，其中有进程间通信、资源共享与竞争（如内存、文件、I/O 访问）、多个进程活动的同步以及分配给进程的处理器时间等。在本章将会看到这些问题不仅会出现在多处理器环境和分布式处理器环境中，也会出现在单处理器的多道程序设计系统中。

3.0 问题导入

当多个进程并发执行时，可能会同时访问一些共享资源，此时如何保证对共享资源的访问是正确有效的呢？例如，有多个售票点同时出售飞机票，如何设计一个正确的并发程序，使得同一航班的同一张票不被出售给多人呢？

3.1 并发概述

3.1.1 并发的概念

我们把系统中可并发执行的进程称为“并发进程”，并发进程相互之间可能是无关的，也可能是有联系的。如果一个进程的执行不影响其他进程的执行，且与其他进程的进展情况不相关，即它们是各自独立的，则称这些并发进程相互之间是无关的。显然，无关的并发进程一定没有共享的变量，它们分别在各自的数据集合上操作。例如，为两个不同源程序进行编译的两个进程可以是并发执行的，但它们之间却是无关的。因为这两个进程分别在不同的数据集合上为不同的源程序进行编译，虽然这两个进程可交叉地占用处理器为各自的源程序进行编译，但是，任何一个进程都不依赖另一个进程。甚至当一个进程发现被编译的源程序有错误时，也不会影响另一个进程继续对自己的源程序进行编译，它们是各自独立的。

然而，如果一个进程的执行依赖其他进程的进展情况，或者说一个进程的执行可能影响其他进程的执行结果，则说这些并发进程相互之间是有交往的、是有关的。例如，有三个进程，即读进程、处理进程和打印进程。其中读进程每次启动外围设备读一批信息并把读入的信息存放到缓冲区，处理进程对存放在缓冲区中的信息加工处理，打印进程把加工处理后的信息打印输出。这三个进程中的每一个进程的执行都依赖另一个进程的进展情况：只有当读进程把一批信息读入并存入缓冲区后，处理进程才能对它进行加工处理，而打印进程要等信息加工处理好后才能把它输出；也只有当缓冲区中的信息被打印进程取走后，读进程才能把读入的第二批信息再存入缓冲区供加工处理；如此循环，直至所有的信息都被处理且

打印输出。可见，这三个进程相互依赖、相互合作，它们是一组有联系的并发进程。有联系的并发进程一定共享某些资源。在上例中从外围设备上读入的信息、经加工处理后的信息、存放信息的缓冲区等都是这组并发进程的共享资源。

3.1.2 时序错误

一个进程运行时由于自身或外界的原因而可能被中断，且断点是不固定的。一个进程被中断后，哪个进程可以运行，被中断的进程什么时候再去占用处理器，这都是与进程调度算法有关的。所以，进程执行的速度不能由自己来控制，对于有联系的并发进程来说，可能有若干并发进程同时使用共享资源，即一个进程一次使用未结束另一个进程就已开始使用，形成交替使用共享资源的现象。如果对这种情况不加控制，就可能出现与时间有关的错误，在共享资源(变量)时就会出错，并得到不正确的结果。请观察下面的例子。


时序错误

例 3-1 飞机售票问题。

假设一个飞机订票系统有两个终端，分别运行进程 T1 和 T2。该系统公共数据区中的一些单元 $Bj(j=1,2,\cdots)$分别存放某日某次航班机票的余数，而 d1 和 d2 表示进程 T1 和 T2 执行时所用的工作单元。飞机售票程序如下。

```
void Ti() (i = 1, 2)
{
   int di = 0;
   [根据旅客订票要求找到 Bj];
   di = Bj ;
   if (di >= 1) {
      di = di - 1;
      Bj = di;
      [打印一张票];
}
   else [提示信息"票已售完"];
}
void main() {
   cobegin
      T1(); T2();
   coend
}
```

由于 T1 和 T2 是两个可同时执行的并发进程，它们在同一个计算机系统中运行，共享同一批票源数据，因此，可能出现如下所示的运行情况(设 $Bj=m$)。

```
T1 : d1 = Bj ;     即 d1 = m ( m > 0 )
T2 : d2 = Bj ;     即 d2 = m
T2 : d2 = d2 - 1 ;
     Bj = d2 ; [打印一张票] ;     即  Bj = m - 1
T1 : d1 = d1 - 1 ;
     Bj = d1 ; [打印一张票] ;     即  Bj = m - 1
```

显然，此时出现了把同一张票卖给了两位旅客的情况，两位旅客都买到一张同天同次航班的机票，可是，Bj 的值实际上只减去了 1，造成余票数的不正确。无论是一票多卖还是余票数量错误，都是不允许的。

例 3-2 主存管理问题。

假定有两个并发进程 Borrow 和 Return 分别负责申请和归还主存资源,在算法描述中,x 表示现有空闲主存总量,B 表示申请或归还的主存量。并发进程算法描述如下。

```
int x = 1000;
void Borrow (int B)
{
   if (B > x) {
      [进程进入等待队列,等待主存资源];
   }
   x = x - B;
   [修改主存分配表,进程获得主存资源];
}
void Return (int B)
{
   x = x + B;
   [修改主存分配表];
   [释放等待主存资源的进程];
}
void main() {
   cobegin
      Borrow(B); Return(B);
   coend
}
```

由于 Borrow 和 Return 共享了表示主存物理资源的临界变量 x,若对并发执行不加限制会导致错误。例如,一个进程调用 Borrow 申请主存,在执行了比较 B 和 x 的指令后,发现 $B>x$,但在执行"进程进入等待队列,等待主存资源"前另一个进程调用 Return 抢先执行,归还了所借全部主存资源。这时,由于申请进程还未成为等待状态,Return 中的"释放等待主存资源的进程"相当于空操作。以后当调用 Borrow 的进程被置成"等待主存资源"时,可能已经没有其他进程来归还主存资源了,从而,申请资源的进程处于永远等待状态。

例 3-3 自动计算问题。

某交通路口设置了一个自动计数系统,该系统由观察者(Observer)进程和报告者(Reporter)进程组成。观察者进程能识别汽车,并对通过的汽车计数,报告者进程定时(可设为每隔一小时)将观察者的计数值打印输出,每次打印后把计数值清"0"。两个进程的并发执行可完成对每小时汽车流量的统计,这两个进程的算法描述如下。

```
int count = 0;
void Observer()
{
   while (true) {
      [observe a car];
      count = count + 1;
   }
}
void Reporter()
{
   while (满足时间间隔) {
      print count;
      count = 0;
   }
}
```

```
void main() {
   cobegin
      Observer(); Reporter();
   coend
}
```

进程 Observer 和 Reporter 并发执行时可能有如下两种情况。

(1) 报告者进程执行时无汽车通过。在这种情况下,报告者进程把上一小时通过的汽车数打印输出后将计数器清"0",完成了一次自己承担的任务。此后,有汽车通过时,观察者进程重新开始对一个新时间段内的流量进行统计。在两个进程的配合下,能正确统计出每小时通过的汽车数量。

(2) 报告者进程执行时有汽车通过。当准点时,报告者进程工作,它启动了打印机,在等待打印机打印输出时恰好有一辆卡车通过,这时,观察者进程占用处理器,把计数器 count 的值又增加了"1"。之后,报告者进程在打印输出后继续执行 count=0。于是,报告者进程在把已打印的 count 值清"0"时,同时把观察者进程在 count 上新增加的"1"也清除了。如果在报告者打印期间连续有车辆通过,虽然观察者都把它们记录到计数器中,但都因报告者执行 count=0 而把计数值丢失了,使统计结果严重失实。

从以上例子可以看出,由于并发进程执行的随机性,一个进程对另一个进程的影响是不可预测的。由于它们共享了资源(变量),当在不同时刻交替访问资源(变量)时就可能造成结果的不正确。造成不正确的因素与进程占用处理器的时间、执行的速度以及外界的影响有关。这些因素都与时间有关,所以,把它们统称为"时序错误"。

3.1.3 临界区

有联系的并发进程执行时出现与时间有关的错误,其根本原因是对共享资源(变量)的使用不加限制,当进程交叉使用了共享资源(变量)时就可能造成错误。为了使并发进程能正确地执行,必须对共享变量的使用加以限制。

我们把并发进程中与共享变量有关的程序段称为"临界区",共享变量所代表的资源称为"临界资源",多个并发进程中涉及相同共享变量的那些程序段称为"相关临界区"。例如,在飞机售票系统中,进程 T1 的临界区为:


临界区

```
d1 = Bj;
if (d1 >= 1) {
    d1 = d1 - 1;
    Bj = d1;
    [打印一张票];
}
else {
   [ 提示信息"票已售完"];
}
```

进程 T2 的临界区为:

```
d2 = Bj;
if (d2 >= 1) {
    d2 = d2 - 1;
    Bj = d2;
    [打印一张票];
```

```
}
else{
    [提示信息"票已售完"];
}
```


这两个临界区都要使用共享变量 Bj,故属于相关临界区。

而在自动计数系统中,观察者进程的临界区是:

```
count = count + 1;
```

报告者进程的临界区是:

```
print count;
count = 0;
```

这两个临界区都要使用共享变量 count,也属于相关区。

如果有进程在相关临界区执行时,不让另一个进程进入相关的临界区执行,就不会形成多个进程对相同共享变量的交叉访问,于是就可避免出现与时间有关的错误。例如,观察者和报告者并发执行时,当报告者启动打印机后,在执行 count=0 之前,虽然观察者发现有汽车通过,应该限制它进入相关临界区(即暂不执行 count=count+1),直到报告者执行了 count=0 退出临界区。当报告者退出临界区后,观察者再进入临界区执行,这样就不会交替地修改 count 值,而观察到的汽车数被统计在下一个时间段内,也不会出现数据丢失。可见,只要对涉及共享变量的临界区互斥执行,就不会出现与时间有关的错误。因而,对若干进程共享某一资源(变量)的相关临界区的管理应满足如下三个要求。

(1) 一次最多让一个进程在临界区执行,当有进程在临界区执行时,其他想进入临界区执行的进程必须等待。

(2) 任何一个进入临界区执行的进程必须在有限的时间内退出临界区,即任何一个进程都不应该无限地逗留在自己的临界区中。

(3) 不能强迫一个进程无限地等待进入它的临界区,即有进程退出临界区时应让一个等待进入临界区的进程进入它的临界区。

3.1.4 进程的互斥

进程的互斥是指当有若干进程都要使用某一共享资源时,任何时刻最多只允许一个进程去使用,其他要使用该资源的进程必须等待,直到占用资源者释放该资源。

实际上,共享资源的互斥使用就是限定并发进程互斥地进入相关临界区。如果能提供一种方法来实现对相关临界区的管理,则就可实现进程的互斥。实现对相关临界区管理的方法有多种,如可采用标志方式、上锁开锁方式、PV 操作方式和管程方式等。

在这里,先介绍几种硬件实现互斥的方案。在这些方案中,当一个进程在临界区中更新共享资源时,其他进程将不会进入其临界区,从而保证程序的正确执行。

1. 中断禁用

在单处理器机器中,并发进程不能重叠执行,只能交替执行。此外,一个进程将一直运行,直到它调用了一个系统服务或被中断。因此为保证互斥,只需要保证一个进程不被中断就可以了,这种能力可以通过系统内核为启用和禁用中断定义的原语来提供。一个进程可以通过下面的方法实施互斥:

```
while (true) {
   /* 禁用中断 */;
   /* 临界区 */;
   /* 启用中断 */;
   /* 其余部分*/;
}
```

由于临界区不能被中断，故可以保证互斥，但是，该方法的代价非常高，由于处理器被限制于只能交替执行程序，因此执行的效率将会有明显的降低。另一个问题是该方法不能用于多处理器结构中，当一个计算机系统包括多个处理器时，就有可能有一个以上的进程同时执行，在这种情况下，禁用中断是不能保证互斥的。

2. 其他处理方法

在多处理器配置中，几个处理器共享内存。在这种情况下，不存在主/从关系，处理器间的行为是无关的，表现出一种对等关系，处理器之间没有支持互斥的中断机制。

在硬件级别上，对存储单元的访问排斥对相同单元的其他访问。基于这一点，处理器的设计者提出了一些机器指令，用于保证两个动作的原子性，如在一个取指令周期中对一个存储器单元的读和写是否唯一。在该指令执行的过程中，任何其他指令访问内存将被阻止，而且这些动作在一个指令周期中完成。

本节给出了两种最常见的指令：比较和交换指令、exchange 指令。

1）比较和交换指令

比较和交换指令定义如下。

```
int compare_and_swap (int *word, int testval, int newval)
{
   int oldval;
   oldval = *word
   if (oldval == testval) *word = newval;
   return oldval;
}
```

该指令的一个版本是用一个测试值(testval)检查一个内存单元(*word)。如果该内存单元的当前值是 testval，就用 newval 取代该值；否则保持不变。该指令总是返回旧内存值，因此，如果返回值与测试值相同，则表示该内存单元已被更新。由此可见这个原子指令由两部分组成：比较内存单元值和测试值；如果值相同，则产生交换(Swap)，整个比较和交换功能按原子操作执行，即它不接受中断。

该指令的另一个版本返回一个布尔(Boolean)值：交换发生时为真(True)；否则为假(False)。几乎所有处理器家族(x86、IA64、SPARC 和 IBMZ 系列机等)中都支持该指令的某个版本，而且多数操作系统都利用该指令支持并发。

图 3-1(a)给出了基于使用这个指令的互斥规程。共享变量 bolt 被初始化为 0。唯一可以进入临界区的进程是发现 bolt 等于 0 的那个进程。所以有试图进入临界区的其他进程进入忙等待模式。术语忙等待(Busy Waiting)或自旋等待(Spin Waiting)指的是这样一种技术：进程在得到临界区访问权之前，它只能继续执行测试变量的指令来得到访问权，除此之外不能做其他事情。当一个进程离开临界区时，它把 bolt 重置为 0，此时只有一个等待进程被允许进入临界区。进程的选择取决于哪个进程正好执行紧接着的比较和交换指令。

```
/* Program mutualexclusion */
const  int   n = /* 进程个数*/;
int bolt;
void P (int i)
{
  while (true) {
    while (compare_and_swap(bolt, 0, 1) == 1)
        /* 不做任何事*/;
    /* 临界区*/;
    bolt = 0;
    /* 其余部分*/;
  }
}
void main()
{
   bolt = 0;
   parbegi n (P(1),P(2),...,P((n));

}
```

(a) 比较和交换指令

```
/* program mutualexclusion */
int const n = /* 进程个数*/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (&keyi, &bolt)
      while (keyi != 0);
      /* 临界区*/;
      bolt = 0;
      /* 其余部分*/;
   }
}
void main()
{
  bolt = 0
  parbegin (P(1),P(2),...,P((n));
}
```

(b) 交换指令

图 3-1　对互斥的硬件支持

2) exchange 指令

exchange 指令定义如下。

```
void exchange (int * register, int * memory)
{
   int temp;
   temp =  * memory;
   * memory =  * register;
   * register = temp;
}
```

该指令交换一个寄存器的内容和一个存储单元的内容。Intel IA-32(Pentium)和 IA-64(Itanium)体系结构都含有 XCHG 指令。

图 3-1(b)显示了基于 exchange 指令的互斥规程：共享变量 bolt 被初始化为 0，每个进程都使用一个局部变量 key 且初始化为 1。唯一可以进入临界区的进程是发现 bolt 等于 0 的那个进程。它通过把 bolt 置为 1 排斥所有其他进程进入临界区。当一个进程离开临界区时，它把 bolt 重置为 0，允许另一个进程进入它的临界区。

注意，由于变量初始化的方式及 exchange 算法的本质，下面的表达式总是成立的：

$$\text{bolt} + \sum_i \text{key}_i = n$$

如果 bolt=0，则没有任何一个进程在它的临界区中；如果 bolt=1，则只有一个进程在临界区中，即 key 的值等于 0 的那个进程。

使用专门的机器指令实施互斥有以下优点。

(1) 适用于在单处理器或共享内存的多处理器上的任何数目的进程。

(2) 非常简单且易于证明。

(3) 可用于支持多个临界区,每个临界区可以用它自己的变量定义。

但是,也有一些严重的缺点。

(1) 使用了忙等待。当一个进程正在等待进入临界区时,它会继续消耗处理器时间。

(2) 可能饥饿。当一个进程离开一个临界区并且有多个进程正在等待时,选择哪一个等待进程是任意的,因此某些进程可能被无限地拒绝进入。

(3) 可能死锁。考虑单处理器中的下列情况。进程 P1 执行专门指令(如 compare&swap、exchange)并进入临界区,然后 P1 被中断并把处理器让给具有更高优先级的 P2。如果 P2 试图使用同一资源,由于互斥机制,它将被拒绝访问。因此,它会进入忙等待循环。但是,由于 P1 比 P2 的优先级低,它将永远不会被调度执行。

3.2 PV 操作

上面所介绍的硬件实现互斥属于忙等待的互斥,本节介绍怎样用 PV 操作来管理相关临界区,亦即用 PV 操作实现进程的互斥。

3.2.1 信号量与 PV 操作


信号量与 PV 操作

1. 信号量

信号量的概念和 PV 操作是荷兰科学家 E. W. Dijkstra 提出来的。信号是交通管理中的一种常用设备,交通管理人员利用信号颜色的变化来实现交通管理。在操作系统中,信号量 S 是一个整数。当 S 大于或等于零时,代表可供并发进程使用的资源实体数;当 S 小于零时,则|S|表示正在等待使用资源实体的进程数。建立一个信号量必须说明此信号量所代表的意义并且赋初值。除赋初值外,信号量仅能通过 PV 操作来访问。

信号量按其用途可分为两种。

(1) 公用信号量,联系一组并发进程,相关的进程均可在此信号量上进行 P 操作和 V 操作,初值常为 1,用于实现进程互斥,也称为互斥信号量。

(2) 私有信号量,联系一组并发进程,仅允许拥有此信号量的进程执行 P 操作,而其他相关进程可在其上施行 V 操作。初值常常为 0 或正整数,多用于实现进程同步,也称为资源信号量。

2. PV 操作

PV 操作是由两个操作,即 P 操作和 V 操作组成。P 操作和 V 操作是两个在信号量上进行操作的过程,假定用 S 表示信号量则把这两个过程记做 P(S)和 V(S),它们的定义如下。

```
void P(Semaphore S)
{
   S = S - 1;
   if (S < 0) Wait(S);
}
void V(Semaphore S)
{
   S = S + 1;
   if (S <= 0) Release(S);
}
```

其中 Wait(S)表示将调用 P(S)过程的进程置成“等待信号量 S”的状态,且将其排入等待队列; Release(S)表示释放一个“等待信号量 S”的进程,使该进程从等待队列退出并加入就绪队列中。

要用 PV 操作来管理共享资源,首先要确保 PV 操作自身执行的正确性。由于 P(S)和 V(S)都是在同一个信号量 S 上操作,为了使得它们在执行时不发生交叉访问信号量 S 而可能出现的错误,约定 P(S)和 V(S)必须是两个不可被中断的过程,即让它们在屏蔽中断下执行。我们把不可被中断的过程称为“原语”,于是 P 操作和 V 操作实际上是“P 操作原语”和“V 操作原语”。在有的教材上 P、V 操作也分别称为 Wait()和 Signal()操作。

P 操作的主要动作如下。

(1) S 减 1。

(2) 若 S 减 1 后仍大于或等于零,则进程继续执行。

(3) 若 S 减 1 后小于零,则该进程被阻塞后放入等待该信号量的等待队列中,然后转进程调度,如图 3-2 所示。

V 操作的主要动作如下。

(1) S 加 1。

(2) 若结果大于零,则进程继续执行。

(3) 若结果小于或等于零,则从该信号的等待队列中释放一个等待进程,然后再返回原进程继续执行或转进程调度,如图 3-3 所示。

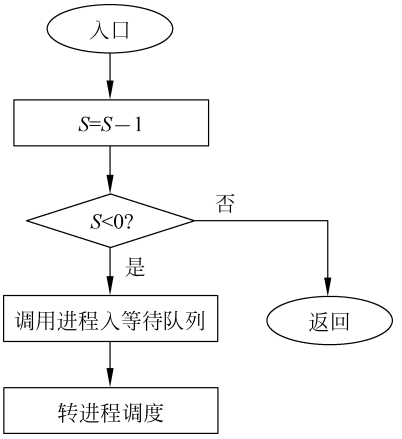

图 3-2 P 操作功能

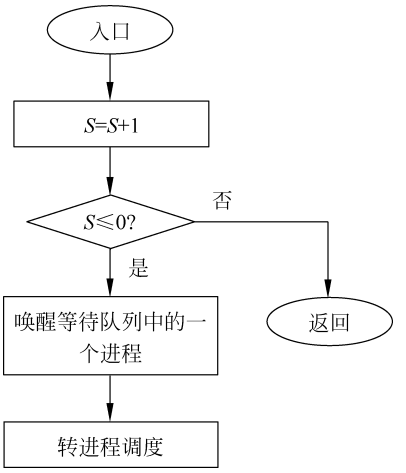

图 3-3 V 操作功能

S 的初值可定义为 0、1 或其他整数,在系统初始化时确定。从信号量和 PV 操作的定义可以获得如下推论。

推论 1: 若信号量 S 为正值,则该值等于 S 所代表的实际可以使用的物理资源数。

推论 2: 若信号量 S 为负值,则其绝对值等于对信号量 S 实施 P 操作而被阻塞并进入信号量 S 等待队列的进程数。

推论 3: 通常 P 操作意味着请求一个资源,V 操作意味着释放一个资源。在一定条件下,P 操作代表阻塞进程操作,而 V 操作代表唤醒被阻塞进程的操作。

3.2.2 用 PV 操作实现进程互斥

用 PV 操作可实现并发进程的互斥，其步骤如下。

(1) 设立一个互斥信号量 S 表示临界区，其取值范围为 1,0,−1,…，其中 $S=1$ 表示无并发进程进入 S 临界区；$S=0$ 表示已有一个并发进程进入 S 临界区；S 等于负数表示已有一个并发进程进入 S 临界区，且有$|S|$个进程等待进入 S 临界区。S 的初值为 1。


用 PV 操作实现进程互斥

(2) 用 PV 操作表示对 S 临界区的申请和释放，在进入临界区之前，通过 P 操作进行申请，在退出临界区之后，通过 V 操作释放。

```
A 进程          B 进程
……              ……
P ( S );        P ( S );
临界区;         临界区;
V ( S );        V ( S );
……              ……
```

下面请看几个实例。

例 3-4 用 PV 操作管理飞机售票问题。

```
Semaphore S;
S = 1;
void Ti() (i = 1, 2)
{
   int di;
   [根据旅客订票要求找到 Bj];
   P(S);
   di = Bj;
   if (di >= 1) {
      di = di − 1;
      Bj = di;
      V(S);
      [输出一张票];
   }
   else {
      V(S);
      [提示信息"票已售完"];
   }
}
void main()
{
   cobegin
      T1(); T2();
   coend
}
```

例 3-5 用 PV 操作管理主存问题。

```
int x;
Semaphore S;
x = 1000; S = 1;
void Borrow (int B)
{
   P(S);
```

```
    if (B > x)  [进程进入等待队列,等待主存资源];
    x = x - B;
    [修改主存分配表,进程获得主存资源];
    V(S);
}
void Return (int B)
{
    P(S);
    x = x + B;
    [修改主存分配表];
    V(S);
    [释放等待主存资源的进程];
}
void main()
{
    cobegin
        Borrow(B); Return(B);
    coend
}
```

例 3-6 用 PV 操作管理自动计数问题。

```
int count;
Semaphore S;
count = 0; S = 1;
void Observer()
{
    while (true) {
        [observe a car];
        P(S);
        count = count + 1;
        V(S);
    }
}
void Reporter()
{
    while (满足时间间隔) {
        P(S);
        print count;
        count = 0;
        V(S);
    }
}
void main() {
    cobegin
        Observer(); Reporter();
    coend
}
```

例 3-7 用 PV 操作解决五个哲学家吃通心面问题。

有五个哲学家围坐在一张圆桌旁,桌子中央有一盘通心面,每人面前有一只空盘子,每两人之间放一根筷子。每个哲学家思考、饥饿,然后欲吃通心面。为了吃面,每个哲学家必须获得两根筷子,且每人只能直接从自己左边或右边取得筷子(如图 3-4 所示)。

这道经典题目中,每一根筷子都是必须互斥使用的,因此,应为每根筷子设置一个互斥

信号量 $Si(i=0,1,2,3,4)$，初值均为 1，当一个哲学家吃通心面之前必须获得自己左边和右边的两根筷子，即执行两个 P 操作；吃完通心面后必须放下筷子，即执行两个 V 操作。

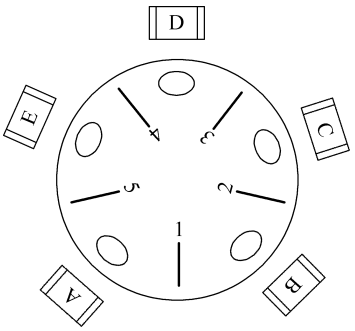

图 3-4　五个哲学家进餐问题

```
Semaphore S0, S1, S2, S3, S4;
S0 = 1; S1 = 1; S2 = 1; S3 = 1; S4 = 1;
void PHi() (i = 0,1,2,3,4)
{
   while (true) {
      [思考];
      P(Si);
      P((Si + 1) % 5);
      [吃通心面];
      V(Si);
      V((Si + 1) % 5);
   }
}
void main() {
   cobegin
      PH0(); PH1(); PH2(); PH3(); PH4();
   coend
}
```

3.3　进 程 同 步

3.3.1　同步的概念

利用信号量解决了进程的互斥问题，但互斥主要是解决并发进程对临界区的使用问题。这种基于临界区控制的交互作用是比较简单的，只要诸进程对临界区的执行时间互斥，每个进程就可忽略其他进程的存在和作用。此外，还需要解决异步环境下的进程同步问题。所谓异步环境，是指相互合作的一组并发进程，其中每一个进程都以各自独立的、不可预知的速度向前推进，但它们又需要密切合作以实现一个共同的任务，即彼此“知道”相互的存在和作用。例如，为了把原始的一批记录加工成当前需要的记录，创建了两个进程，即进程 A 和进程 B。进程 A 启动输入设备不断地读记录，每读出一个记录就交给进程 B 去加工，直至所有记录都处理结束。为此，系统设置了一个容量为存放一个记录的缓冲器，进程 A 把读出的记录存入缓冲器，进程 B 从缓冲器中取出记录加工，如图 3-5 所示。

图 3-5　进程协作

进程 A 和进程 B 是两个并发进程，它们共享缓冲器，如果两个进程不相互制约就会造成错误。当进程 A 的执行速度超过进程 B 的执行速度时，可能进程 A 把一个记录存入缓冲器后，在进程 B 还没有取走前，进程 A 又把新读出的一个记录存入缓冲器，后一个记录就把前一个尚未取走的记录覆盖了，造成记录的丢失。当进程 B 的执行速度超过进程 A 的执行

速度时,可能进程B从缓冲器取出一个记录并加工后,进程A还没有把下一个新记录存入缓冲器,而进程B却又从缓冲器中去取记录,造成重复地取同一个记录加工。

用进程互斥的办法不能克服上述两种错误,事实上,进程A和进程B虽然共享缓冲器,但它们都是在无进程使用缓冲器时才向缓冲器存记录或从缓冲器取记录的。也就是说,它们在互斥使用共享缓冲器的情况下仍会发生错误,引起错误的根本原因是它们之间的相对速度。可以采用互通消息的办法来控制执行速度,使相互协作的进程正确工作。

两个进程应该按照如下原则协作。

(1) 进程A把一个记录存入缓冲区后,应向进程B发送"缓冲器中有等待处理的记录"的消息。

(2) 进程B从缓冲器中取出记录后,应向进程A发送"缓冲器中的记录已取走"的消息。

(3) 进程A只有在得到进程B发送来的"缓冲器中的记录已取走"的消息后,才能把下一个记录再存入缓冲器。否则进程A等待,直到消息到达。

(4) 进程B只有在得到进程A发送来的"缓冲器中有等待处理的记录"的消息后,才能从缓冲器中取出记录并加工。否则进程B等待,直到消息到达。

由于每个进程都是在得到对方的消息后才去使用共享的缓冲器,所以不会出现记录的丢失和记录的重复处理。

因此,进程的同步是指导并发进程之间存在一种制约关系,一个进程的执行依赖另一个进程的消息,当一个进程没有得到另一个进程的消息时应等待,直到消息到达时才被唤醒。

3.3.2 PV操作实现进程同步

要实现进程的同步就必须提供一种机制,该机制能把其他进程需要的消息发送出去,也能测试自己需要的消息是否到达。把能实现进程同步的机制称为同步机制,不同的同步机制实现同步的方法也不同,PV操作和管程是两种典型的同步机制。本节介绍怎样用PV操作实现进程间的同步。

我们已经知道怎样用PV操作来实现进程的互斥。事实上,PV操作不仅是实现进程互斥的有效工具,而且还是一个简单而方便的同步工具。用一个信号量与一个消息联系起来,当信号量的值为"0"时表示期望的消息尚未产生,当信号量的值为非"0"时表示期望的消息已经存在。假定用信号量S表示某个消息,现在来看看怎样用PV操作达到进程同步的目的。

1. 调用P操作测试消息是否到达

任何进程调用P操作可测试到自己所期望的消息是否已经到达。若消息尚未产生则$S=0$,调用P(S)后,P(S)一定让调用者成为等待信号量S的状态,即调用者此时必定等待直到消息到达;若消息已经存在则$S\neq0$,调用P(S)后进程不会成为等待状态而可继续执行,即进程测试到自己期望的消息已经存在。

2. 调用V操作发送消息

任何进程要向其他进程发送消息时可调用V操作。若调用V操作之前$S=0$,表示消息尚未产生且无等待消息的进程,这时调用V(S)后执行$S=S+1$使$S\neq0$,即意味着消息已存在;若调用V操作之前$S<0$,表示消息未产生前已有进程在等待消息,这时调用

V(S)后将释放一个等待消息者，即表示该进程等待的消息已经到达可以继续执行。

在用 PV 操作实现同步时，一定要根据具体的问题来定义信号量和调用 P 操作或 V 操作。一个信号量与一个消息联系在一起，当有多个消息时必须定义多个信号量；测试不同的消息是否到达或发送不同的消息时，应对不同的信号量调用 P 操作或 V 操作。

3.3.3 生产者-消费者问题


生产者-消费者问题

生产者-消费者问题是一个典型的同步例子。假定有一个生产者和一个消费者，他们共用一个缓冲器，生产者不断地生产物品，每生产一件物品就要存入缓冲器，但缓冲器中每次只能存入一件物品，只有当消费者把物品取走后，生产者才能把下一件物品存入缓冲器。同样地，消费者要不断地从缓冲器取出物品消费，当缓冲器中有物品时他就可以去取，每取走一件物品后必须等生产者再放一件物品后才能再取。

在这个问题中，生产者要向消费者发送“缓冲器中有物品”的消息，而消费者要向生产者发送“可把物品存入缓冲器”的消息。用 PV 操作实现生产者-消费者之间的同步，应该定义两个信号量，分别表示两个消息。我们把这两个信号量定义为 sPdt 和 sGet，它们的含义如下。

(1) sPdt。表示是否可以把物品存入缓冲器，由于缓冲器中只能放一件物品，系统初始化时应允许放入物品，所以 sPdt 的初值应为“1”。

(2) sGet。表示缓冲器中是否存有物品，显然，系统初始化时缓冲器中应该无物品，所以 sGet 的初值应为“0”。

对生产者来说，生产一件物品后应调用 P(sPdt)，当缓冲器中允许放物品时(sPdt = 1)，则在调用 P(sPdt)后可以把物品存入缓冲器(此时 sPdt 的值已变为 0)。生产者把一件物品存入缓冲器后，又可继续去生产物品，但若消费者尚未取走上一件物品(这时 sPdt 维持为 0)，而生产者欲把生产的物品存入缓冲器时调用 P(sPdt)后将成为等待状态，阻止它把物品存入缓冲器。生产者在缓冲器中每存入一件物品后，应调用 V(sGet)把缓冲器中有物品的消息告诉消费者(调用 V(sGet)后，sGet 的值从 0 变为 1)。

对消费者来说，取物品前应查看缓冲器中是否有物品，即调用 P(sGet)。若缓冲器中尚无物品(sGet 仍为 0)，则调用 P(sGet)后消费者等待，不能去取物品，直到生产者存入一件物品后发送有物品的消息时才唤醒消费者。若缓冲器中已有物品(sGet 为 1)，则调用 P(sGet)后消费者可继续执行，从缓冲器中去取物品。消费者从缓冲器中每取走一件物品后应调用 V(sPdt)，通知生产者缓冲器中物品已取走，可以存入一件新物品。

例 3-8 生产者和消费者并发执行时，用 PV 操作作为同步机制可按如下方式管理。

```
Semaphore sPdt, sGet;
int Buffer;
sPdt = 1; sGet = 0;
void Producer()
{
   while (true) {
      [Produce a product];
      P(sPdt);
      buffer = product;
      V(sGet);
```

```
    }
}
void Consumer()
{
    while (true) {
        P(sGet);
        [Take a product from buffer];
        V(sPdt);
        [Consume];
    }
}
void main() {
    cobegin
        Producer(); Consumer();
    coend
}
```

请注意,生产者生产物品的操作和消费者消费物品的操作是各自独立的,只是在访问公用的缓冲器把物品存入或取出时才要互通消息。所以,测试消息是否到达和发送消息的P操作与V操作应该分别在访问共享缓冲器之前和之后。

如果一个生产者和一个消费者共享的缓冲器容量为可以存放 n 件物品($n>1$),那么只要把信号量 sPdt 的初值定为 n,sGet 的初值仍为"0"。当缓冲器中没有放满 n 件物品时,生产者调用 P(sPdt)后都不会成为等待状态而可以把生产出来的物品存入缓冲器。但当缓冲器中已经有 n 件物品时(sPdt 值为 0),生产者再想存入一件物品将被拒绝。生产者每存入一件物品后,由于调用 V(sGet)发送消息,故 sGet 的值表示缓冲器中可供消费的物品数。只要 sGet≠0,消费者调用 P(sGet)后总可以去取物品,每取走一件物品后调用 V(sPdt),便增加了一个可以用来存放物品的位置。

由于缓冲器可存 n 件物品,因此,必须指出缓冲器中什么位置已有物品可供消费,什么位置尚无物品可供生产者存放物品。可以用两个指针 pp 和 cp 分别指示生产者往缓冲器存物品与消费者从缓冲器取物品的相对位置,它们的初值为 0,生产者和消费者按顺序的位置去存物品和取物品。缓冲器被循环使用,即生产者在缓冲器顺序存放了 n 件物品后,则以后继续生产的物品仍从缓冲器的第一个位置开始存放。于是,一个生产者和一个消费者共享容量为 n 的缓冲器时,可如下进行同步工作。

```
int Buf[ n ];
Semaphore sPdt, sGet;
int pp, cp;
sPdt = n; sGet = 0; pp = 0; cp = 0;
void Producer()
{
    while (true) {
        [Produce a product];
        P(sPdt);
        Buf[pp] = product;
        pp = (pp + 1) % n;
        V(sGet);
    }
}
void Consumer()
```

```
{
   while (true) {
      P(sGet);
      [Take a product from Buf[cp] ];
      cp = (cp + 1) % n;
      V(sPdt);
      [consume];
   }
}
void main() {
   cobegin
      Producer(); Consumer();
   coend
}
```

但是，要提醒注意的是，如果 PV 操作使用不当，仍会出现与时间有关的错误。例如，有 p 个生产者和 q 个消费者，它们共享可存放 n 件物品的缓冲器；为了使它们能协调工作，必须使用一个互斥信号量 S(初值为 1)，以限制它们对缓冲器互斥地存取；另外，使用两个信号量 sPdt(初值为 n)和 sGet(初值为 0)来保证生产者不往满的缓冲器中存放物品，消费者不从空的缓冲器中取出物品。同步工作描述如下。

```
int Buf[ n ];
Semaphore sPdt, sGet, S;
int pp, cp;
sPdt = n; sGet = 0; pp = 0; cp = 0; S = 1;
void Producer_i() (i = 1, 2, …, p)
{
   while (true) {
      [Produce a product];
      P(sPdt);
      P(S);
      Buf[pp] = product;
      pp = (pp + 1) % n;
      V(sGet);
      V(S);
   }
}
void Consumer_j() (j = 1, 2, …, q)
{
   while (true) {
      P(sGet);
      P(S);
      [Take a product from Buf[cp] ];
      cp = (cp + 1) % n;
      V(sPdt);
      V(S);
      [consume];
   }
}
void main() {
   cobegin
      Producer_i() (i = 1, 2, …, p); Consumer_j() (j = 1, 2, …, q);
   coend
}
```

在这个例子中,P操作的顺序是很重要的,如果把生产者和消费者进程中的两个P操作交换顺序,则会导致错误。而V操作的顺序却是无关紧要的。一般来说,用于同步的信号量上的P操作在前执行,而用于互斥的信号量上的P操作在后执行。

生产者-消费者问题是非常典型的问题,有许多问题可归结为生产者-消费者问题,但要根据实际情况灵活运用。例如,现有四个进程R1、R2、P1、P2,它们共享可以存放一个数的缓冲器Buf。进程R1每次把来自键盘的一个数存入缓冲器Buf中,供进程P1打印输出;进程R2每次从磁盘上读一个数存放到缓冲器Buf中,供进程P2打印输出。为防止数据的丢失和重复打印,怎样用PV操作来协调这四个进程的并发执行?

先来分析一下这四个进程的关系,进程R1和进程R2相当于两个生产者,接收来自键盘的数或从磁盘上读出的数相当于这两个进程各自生产的物品。两个进程各自生产的不同物品要存入共享的缓冲器Buf中,由于Buf中每次只能存入一个数,因此进程R1和进程R2在存数时必须互斥。进程P1和进程P2相当于两个消费者,它们分别消费进程R1和进程R2生产的物品。所以进程R1(或进程R2)在把数存入缓冲器Buf后应发送消息通知进程P1(或进程P2)。进程P1(或进程P2)在取出数之后应发送消息通知进程R1(或进程R2)告知缓冲器中又允许放一个新数的消息。显然,进程R1与进程P1、进程R2与进程P2之间要同步。

在分析了进程之间的关系后,应考虑怎样来定义信号量。首先,应定义一个是否允许进程R1或进程R2把数存入缓冲器的信号量S,其初值为1。其次,进程R1或进程R2分别要向进程P1和进程P2发送消息,应该要有两个信号量$S1$和$S2$来表示相应的消息,初值都应为0,表示缓冲器中尚未有数。至于进程P1或进程P2从缓冲器中取出数后要发送“缓冲器中允许放一个新数”的消息,这个消息不应该特定地发给进程R1或进程R2,所以只要调用V(S)就可达到目的。到底哪个进程可以把数存入缓冲器中,由进程R1或进程R2调用P(S)来竞争。因此,不必再增加新信号量了。现定义三个信号量,其物理含义如下。

S:表示能否把数存入缓冲器Buf。

$S1$:表示缓冲器中是否存有来自键盘的数。

$S2$:表示缓冲器中是否存有从磁盘上读取的数。

例3-9 四个进程可如下协调工作。

```
int Buf;
Semaphore S, S1, S2;
S = 1; S1 = 0; S2 = 0;
void R1()
{
   int x;
   while (true) {
      [接收来自键盘的数];
      x = 接收的数;
      P(S);
      Buf = x;
      V(S1);
   }
}
void R2()
```

```
{
    int y;
    while (true) {
        [从磁盘上读一个数];
        y = 读入的数;
        P(S);
        Buf = y;
        V(S2);
    }
}
void P1()
{
    int k;
    while (true) {
        P(S1);
        k = Buf;
        V(S);
        [打印 k 的值];
    }
}
void P2()
{
    int j;
    while (true) {
        P(S2);
        j = Buf;
        V(S);
        [打印 j 的值];
    }
}
void main() {
    cobegin
        R1(); R2(); P1(); P2();
    coend
}
```

在这里，进程 R1 和进程 R2 在向缓冲器 Buf 中存数之前调用了 P(S)，其有两个作用。

(1) 由于 S 的初值为 1，所以 P(S)限制了每次至多只有一个进程可以向缓冲器中存入一个数，起到了互斥地向缓冲器中存数的作用。

(2) 当缓冲器中有数且尚未被取走时 S 的值为 0，当缓冲器中数被取走后 S 的值又为 1，因此 P(S)起到了测试“允许存入一个新数”的消息是否到达的同步作用。

进程 P1 和进程 P2 把需要的数取走后，都调用 V(S)发出可以存放一个新数的消息。可见，在这个问题中信号量 S 既被作为互斥的信号量，又被作为同步的信号量。

在操作系统中进程同步问题是非常重要的，通过对一些例子的分析大家应该学会怎样区别进程的互斥和进程的同步。PV 操作是实现进程互斥和进程同步的有效工具，但若使用不得当则不仅会降低系统效率而且仍会产生错误，希望读者在弄清 PV 操作作用的基础上，体会在各个例子中调用不同信号量上的 P 操作和 V 操作的目的，从而正确掌握对各类问题的解决方法。

3.3.4 读者-写者问题

读者-写者问题也是一个经典的并发程序设计问题。有两组并发进程：读者和写者共享一个文件 F,要求：①允许多个读者同时对文件执行读操作；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前,应让已有的写者和读者全部退出。


读者-写者问题

单纯使用信号量不能解决读者-写者问题,必须引入计数器 rc 记录读进程数,rmutex 是用于对计数器 rc 操作的互斥信号量,*W* 表示是否允许写的信号量,于是管理该文件的同步工作描述如下。

例 3-10 读者-写者进程同步操作。

```
Semaphore rmutex, W;
int rc;
rmutex = 1; rc = 0; W = 1;
void Reader_i() ( i = 1, 2, … )
{
   P (rmutex);
   rc = rc + 1;
   if (rc == 1) P (W);
   V (rmutex);
   [读文件];
   P (rmutex);
   rc = rc - 1;
   if (rc == 0) V (W);
   V (rmutex);
}
void Writer_j() ( j = 1, 2, … )
{
   P (W);
   [写文件];
   V (W);
}
void main() {
   cobegin
      Reader_i() ( i = 1, 2, … ); Writer_j() ( j = 1, 2, … );
   coend
}
```

在上面的方法中,读者是优先的。当存在读者时,写操作将被延迟,并且只要有一个读者在访问文件,随后而来的读者都将被允许访问文件。从而导致了写进程长时间等待,并有可能出现写进程被“饿死”。增加信号量并修改上述程序可以得到写进程具有优先权的解决方案能保证当一个写进程声明想写时,不允许新的读进程再访问共享文件。

对于写进程在已有定义的基础上还必须增加下列信号量和变量,引入计数器 wc 记录写进程数,wmutex 是用于对计数器 wc 操作的互斥信号量。*R* 表示是否允许读的信号,当至少有一个写进程准备访问文件时,用于禁止所有的读进程。

对于读进程还需要一个额外的信号量。在 *R* 上不允许建造长队列,否则写进程将不能跳过这个队列,因此,只允许一个读进程在 *R* 上排队,而所有其他读进程在等待 *R* 之前,在信号量 rlist 上排队。

例 3-11 写者优先，读者-写者进程同步操作。

```
Semaphore rmutex, wmutex, rlist, W, R;
int rc, wc;
rmutex = 1; wmutex = 1; rlist = 1; W = 1; R = 1; rc = 0; wc = 0;
void Reader_i() ( i = 1, 2, … )
{
   P (rlist);
   P (R);
   P (rmutex);
   rc = rc + 1;
   if (rc == 1) P (W);
   V (rmutex);
   V (R);
   V (rlist);
   [读文件];
   P (rmutex);
   rc = rc − 1;
   if (rc == 0) V (W);
   V (rmutex);
}
void Writer_j() ( j = 1, 2, … )
{
   P (wmutex);
   wc = wc + 1;
   if (wc == 1) P(R);
   V(wmutex);
   P (W);
   [写文件];
   V (W);
   P (wmutex);
   wc = wc − 1;
   if (wc == 0) V(R);
   V(wmutex);
}
void main() {
   cobegin
      Reader_i() ( i = 1, 2, … ); Writer_j() ( j = 1, 2, … );
   coend
}
```

3.3.5 时间同步问题

前面讲到的进程同步都属于空间上的同步问题，其实进程同步还有个时间上的同步问题。当一组有关的并发进程在执行时间上有严格的先后顺序时，就会出现时间上的进程同步问题。例如，有 7 个进程，它们的执行顺序如图 3-6 所示。

时间同步问题

为了保证这 7 个进程严格按照顺序执行，可定义 6 个信号量，其物理含义如下。

S2：表示进程 P2 能否执行。

S3：表示进程 P3 能否执行。

S4：表示进程 P4 能否执行。

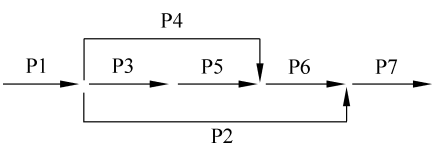

图 3-6 7 个进程的执行顺序

S5：表示进程 P5 能否执行。

S6：表示进程 P6 能否执行。

S7：表示进程 P7 能否执行。

进程 P1 不需定义信号量，可随时执行。这些信号量的初值为 0，表示不可执行，而当信号量大于或等于 1 时，表示可执行。

例 3-12 进程执行顺序同步工作描述如下。

```
Semaphore S2, S3, S4, S5, S6, S7;
S2 = 0; S3 = 0; S4 = 0;
S5 = 0; S6 = 0; S7 = 0;
void P1()
{
   …
   V (S2);
   V (S3);
   V (S4);
}
void P2()
{
   P (S2);
   …
   V (S7);
}
void P3()
{
   P (S3);
   …
   V (S5);
}
void P4()
{
   P (S4);
   …
   V (S6);
}
void P5()
{
   P (S5);
   …
   V (S6);
}
void P6()
{
   P (S6);
   P (S6);
   …
   V (S7);
}
void P7()
{
   P (S7);
   P (S7);
   …
}
void main() {
   cobegin
      P1(); P2(); P3(); P4();
      P5(); P6(); P7();
   coend
}
```

当 P1 执行完后，执行了 V(S2)、V(S3)和 V(S4)三个 V 操作，使 P2、P3 和 P4 在 P1 后可并发执行。P3 执行完后，执行了 V(S5)操作，则可启动 P5 执行。而 P6 要等 P4 与 P5 两个进程全部执行完，执行了两个 V(S6)操作后，才能启动执行。P7 要等 P2 与 P6 两个进程全部执行完，执行了两个 V(S7)操作后才能启动执行。这样，就可以保证 7 个进程在时间上的同步。

3.4 管　　程

3.4.1 什么是管程

信号量机制为实现进程的同步与互斥提供一种原始、功能强大且灵活的工具，然而在使用信号量和 PV 操作实现进程同步时，对共享资源的管理分散于各个进程中，进程能够直接对共享变量进行处理，这样不利于系统对临界资源的管理，难以防止进程有意或无意地违反

同步操作，且容易造成程序设计错误。因此，在进程共享主存的前提下，如果能集中和封装针对一个共享资源的所有访问并包括所需的同步操作，即把相关的共享变量及其操作集中在一起统一控制和管理，就可以方便地管理和使用共享资源，使并发进程之间的相互作用更为清晰，也更易于编写正确的并发程序。

1973 年，Hansen 和 Hoare 正式提出了管程(monitor)的概念，并对其做了如下的定义：关于共享资源的数据及在其上操作的一组过程或共享数据结构及其规定的所有操作。管程的引入可以让我们按资源管理的观点，将共享资源和一般资源管理区分开来，使进程同步机制的操作相对集中。采用这种方法，对共享资源的管理可借助数据结构及其上所实施操作的若干进程来进行；对共享资源的申请和释放可通过进程在数据结构上的操作来实现。管程被请求和释放资源的进程所调用，管程实质上是把临界区集中到抽象数据类型模板中，可作为程序设计语言的一种结构成分。对于同步问题的解决，管程和信号量具有同等的表达能力。

3.4.2 使用信号量的管程

管程是由一个或多个过程、一个初始化序列和局部数据组成的软件模块，其主要特点如下。

(1) 局部数据变量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) 在任何时候，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阻塞，以等待管程可用。


使用信号量的管程

前两个特点让人联想到面向对象软件中对象的特点。的确，面向对象操作系统或程序设计语言可以很容易地把管程作为一种具有特殊特征的对象来实现。

通过给进程强加规定，管程可以提供一种互斥机制：管程中的数据变量每次只能被一个进程访问到。因此，可以把一个共享数据结构放在管程中，从而提供对它的保护。如果管程中的数据代表某些资源，那么管程为访问这些资源提供了互斥机制。

为进行并发处理，管程必须包含同步工具。例如，假设一个进程调用了管程，并且当它在管程中时必须被阻塞，直到满足某些条件。这就需要一种机制，使得该进程不仅被阻塞，而且能释放这个管程，以便某些其他的进程可以进入。以后，当条件满足且管程再次可用时需要恢复该进程并允许它在阻塞点重新进入管程。

管程通过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。有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 上阻塞，管程现在可被另一个进程使用。

(2) csignal(c)。恢复执行在 cwait 之后因为某些条件而阻塞的进程。如果有多个这样的进程，选择其中一个；如果没有这样的进程，什么也不做。

注意，管程的 wait 和 signal 操作与信号量不同。如果在管程中的一个进程发信号，但没有在这个条件变量上等待的任务，则丢弃这个信号。

图 3-7 给出了一个管程的结构。尽管一个进程可以通过调用管程的任何一个过程进入管程，但我们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被阻塞并加入等待管程可用的进程队列中。当一个进程在管程中

时,它可能会通过发送 cwait(x)把自己暂时阻塞在条件 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,在 cwait(x)调用的下一条指令开始恢复执行。

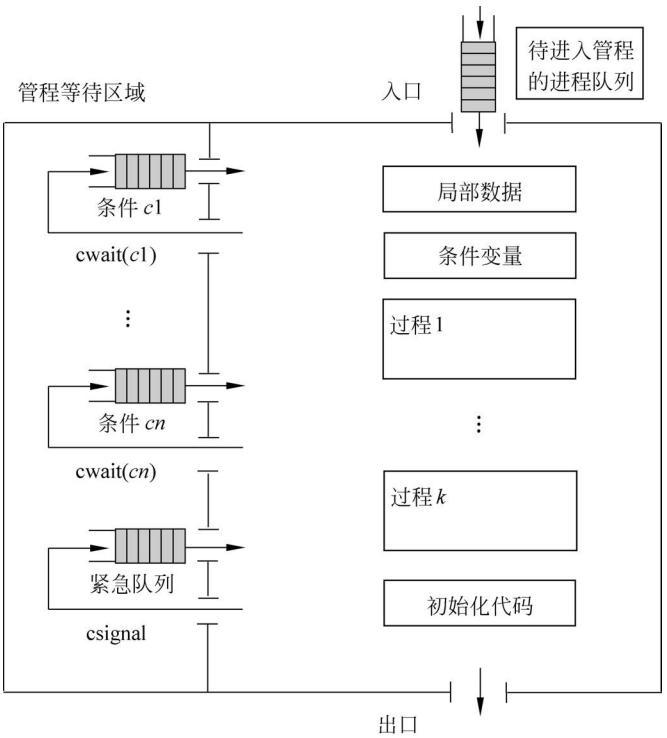

图 3-7　管程的结构

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,它将发送 csignal(x),通知相应的条件队列条件已改变。

为给出一个使用管程的例子,我们再次考虑有界缓冲区的生产者/消费者问题。例 3-13 给出了使用管程的一种解决方案,管程模块 PC 控制着用于保存和取回物品的缓冲区,管程中有两个条件变量(使用结构 condition 声明):当缓冲区中至少有增加一个物品的空间时,notFull 为真;当缓冲区中到少有一个物品时,notEmpty 为真。

生产者可以通过管程中的过程 put 往缓冲区中存放物品,它不能直接访问 buffer。该过程首先检查条件 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阻塞。其他某个进程(生产者或消费者)现在可以进入管程。后来,当缓冲区不再满时,被阻塞进程可以从队列中移出,重新被激活,并恢复处理。在往缓冲区中放置一个物品后,该进程发送 notEmpty 条件信号。对消费者函数也可以进行类似的描述。

这个例子指出,与信号量相比较,管程担负的责任不同。对于管程,它构造了自己的互斥机制:生产者和消费者不可能同时访问缓冲区;但是,程序员必须把适当的 cwait 和 csignal 原语放在管程中,用于防止进程往一个满缓冲区中存放数据项,或者从一个空缓冲区中取数据项。而在使用信号量的情况下,执行互斥和同步都属于程序员的责任。

例 3-13　PC 管程可描述如下。

```
Monitor producer - consumer{
  char buffer[ n ];                                  /* 分配 n 个字符型数据空间 */
```

```
  int nextIn, nextOut;                              /* 缓冲区指针 */
  int count;                                        /*缓冲区中数据项的个数*/
  condition notFull, notEmpty;                      /*为同步设置的条件变量*/
  void put (char x)
  {
       if (count >= n) cwait(notFull);              /*缓冲区满,防止溢出*/
       buffer[nextIn] = x;
       nextIn = (nextIn + 1) % n;
       count++;                                     /*缓冲区中数据项个数增一*/
       csignal(notEmpty);                           /*释放任何一个等待的进程*/
  }
  void take (char x)
  {
       if (count <= 0) cwait(notEmpty);             /*缓冲区空,防止下溢*/
       x = buffer[nextOut];
       nextOut = (nextOut + 1) % n;
       count--;                                     /*缓冲区中数据项个数减一*/
       csignal (notFull);                           /* 释放任何一个等待的进程*/
  }
  {   nextIn = 0; nextOut = 0; count = 0; }         /*缓冲区初始化为空*/
}PC
```

在利用管程解决生产者-消费者问题时,其中的生产者和消费者可描述如下。

```
void Producer()
{
    char x;
    while (true) {
        produce(x);
        PC.put(x); }
}
void Consumer()
{
    char x;
    while (true) {
        PC.take(x);
        consume (x); }
}
void main() {
   parbegin (Producer, Consumer);
}
```

注意,在上述程序中,进程在执行 csignal 函数后立即退出管程,如果在过程最后没有发生 csignal,Hoare 建议发送该信号的进程被阻塞,从而使管程可用,并被放入队列中直到管程空闲。此时一种可能是把阻塞进程放置到入口队列中,这样它就必须与其他还没有进入管程的进程竞争。但是,由于在 csignal 函数上阻塞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,如图 3-7 所示。

如果没有进程在条件 x 上等待,那么 csignal(x)的执行将不会产生任何效果。而对于信号量,在管程的同步函数中可能会产生错误。例如,如果省略掉 PC 管程中的任何一个 csignal 函数,那么进入相应条件队列的进程将被永久阻塞。管程优于信号量之处在于,所有的同步机制都被限制在管程内部,因此,不但易于验证同步的正确性,而且易于检测出错

误。此外,如果一个管程被正确地编写,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被正确地编写时,资源访问才是正确的。

3.4.3 使用通知和广播的管程

Hoare关于管程的定义要求在条件队列中至少有一个进程,当另一个进程为该条件产生csignal时,该队列中的一个进程立即运行。因此产生csignal的进程必须立即退出管程,或者阻塞在管程上。

使用通知和广播的管程

这种方法有两个缺陷。

(1) 如果产生csignal的进程在管程内还未结束,则需要两个额外的进程切换:阻塞这个进程需要一次切换,当管程可用时恢复这个进程又需要一次切换。

(2) 与信号相关的进程调度必须非常可靠。产生一个csignal时,来自相应条件队列中的一个进程必须立即被激活,调度程序必须确保在激活前没有其他进程进入管程,否则,进程被激活的条件又会改变。例如,在例3-13中,当产一个csignal(notEmpty)时,来自notEmpty队列中的一个进程必须在一个新消费者进入管程之前被激活。另一个例子是,生产者进程可能往一个空缓冲区中添加一个字符,并在发信号之前失败,那么在notEmpty队列中的任何进程都将被永久阻塞。

Lampson和Redell为Mesa语言开发了一种不同的管程,他们的方法克服了上面列出的问题,并支持许多有用的扩展,Mesa管理结构还可以用于Modula-3系统程序设计语言。在Mesa中,csignal原语被cnotify取代,cnotify可解释如下:当一个正在管程中的进程执行cnotify(x)时,它使得x条件队列得到通知,但发信号的进程继续执行。通知的结果是使得位于条件队列头的进程在将来合适的时候且当处理器可用时被恢复执行。但是,由于不能保证在它之前没有其他进程进入管程,因而这个等待进程必须重新检查条件。例如,PC管程中的过程现在采用如例3-14所示的代码。

例3-14 使用通知和广播的PC管程描述如下。

```
void put (char x)
{
     while (count >= n) cwait (notFull);          /* 缓冲区满,防止溢出 */
     buffer[nextIn] = x;
     nextIn = (nextIn +1) % n;
     count++;                                     /* 缓冲区中数据项个数增一 */
     cnotify(notEmpty);                           /* 通知正在等待的进程 */
}
void take (char x)
{
     while (count <= 0) cwait(notEmpty);          /* 缓冲区空,防止下溢 */
     x = buffer[nextOut];
     nextOut = (nextOut + 1) % n;
     count--;                                     /* 缓冲区中数据项个数减一 */
     cnotify(notFull);                            /* 通知正在等待的进程 */
}
```

if语句被while循环取代,因此,这个方案导致对条件变量至少多一次额外的检测。作为回报,它不再有额外的进程切换,并且对等待进程在cnotify之后什么时候运行没有任何限制。

与 cnotify 原语相关的一个很有用的改进是，给每个条件原语关联一个监视计时器，不论条件是否被通知，一个等待时间超时的进程将被设置为就绪态。当被激活后，该进程检查相关条件，如果条件满足则继续执行。超时可以防止如下情况的发生：当某些其他进程在产生相关条件的信号之前失败时，等待该条件的进程被无限制地推迟执行而处于“饥饿”状态。

由于进程是接到通知而不是强制激活的，因此就可以给指令表中增加一条 cbroadcast 原语。广播可以使所有的该条件上等待的进程都被置于就绪态，当一个进程不知道有多少进程将被激活时，这种方式是非常方便的。例如，在生产者、消费者问题中，假设 put 和 take 函数都适用于可变长度的字符块，此时，如果一个生产者往缓冲区中添加一批字符，它不需要知道每个正在等待的消费者准备消耗多少字符，而仅仅产生一个 cbroadcast，所有正在等待的进程都得到通知并再次尝试运行。

此外，当一个进程难以准确地判定将激活哪个进程时，也可使用广播。存储管理程序就是一个很好的例子。管理程序有 j 个空闲字节，一个进程释放了额外的 k 个字节，但它不知道哪个等待进程一共需要 $k+j$ 个字节，因此它使用广播，所有进程都检测是否有足够的存储空间。

Lampson/Redell 管程优于 Hoare 管程之处在于，Lampson/Redell 方法的错误比较少。在 Lampson/Redell 方法中，由于每个过程在收到信号后都检查管程变量，且由于使用了 while 结构，一个进程不正确的广播会发信号，不会导致收到信号的程序出错。收到信号的程序将检查相关的变量，如果期望的条件不满足，它会继续等待。

Lampson/Redell 管程的另一个优点是，它有助于在程序结构中采用更模块化的方法。例如，考虑一个缓冲区分配程序的实现，为了在顺序的进程间合作，必须满足以下两级条件。

(1) 保持一致的数据结构。管程强制实施互斥，并允许对缓冲区的另一个操作之前完成一个输入或输出操作。

(2) 在 1 级条件的基础上，加上完成该进程请求，分配给该进程所需的足够存储空间。

在 Hoare 管程中，每个信号传达 1 级条件，同时出携带一个隐含消息，“我现在有足够的空闲字节，能够满足特定的分配请求”，因此该信号隐式携带 2 级条件。如果后来程序员改变了 2 级条件的定义，则需要重新编写所有发信号的进程；如果程序员改变了对任何特定等待进程的假设(也就是说，等待一个稍微不同的 2 级不变量)，则可能需要重新编写所有发信号的进程。这样就不是模块化的结构，并且当代码被修改后可能会引发同步错误(如被错误条件唤醒)。每当对 2 级条件做很小的改动时，程序员必须记得去修改所有的进程。而对于 Lampson/Redell 管程，一次广播可以确保 1 级条件并携带 2 级条件的线索，每个进程将自己检查 2 级条件。不论是等待者还是发信号者对 2 级条件进行了改动，由于每个过程都会检查自己的 2 级条件，故不会产生错误的唤醒。因此，2 级条件可以隐藏在每个过程中。而对 Hoare 管程，2 级条件必须由等待者带到每个发信号的进程的代码中，这违反了数据抽象和进程间的模块化原则。

3.4.4 用管程解决哲学家进餐问题

现在介绍如何用管程来解决哲学家进餐问题。在这里，认为哲学家可以处在这样三种

状态之一：即进餐、饥饿和思考。相应地，引入数据结构：

```
(thinking, hungry, eating) state[5];
```

为每一位哲学家设置一个条件变量 self(i)，每当哲学家饥饿但又不能获得进餐所需的筷子时，他可以执行 cwait(self(i))操作，来推迟自己进餐。条件变量可描述为：

```
condition self[5];
```

在管程中还设置了以下三个过程。

(1) pickup(i:0..4)过程。在哲学家进程中，可利用该过程去进餐。如某哲学家是处于饥饿状态，且他的左、右两个哲学家都未进餐时，便允许这位哲学家进餐，因为他此时可以拿到左、右两根筷子；但只要其左、右两位哲学家中有一位正在进餐时，便不允许该哲学家进餐，此时将执行 cwait(self(i))操作来推迟进餐。

(2) putdown(i:0..4)过程。当哲学家进餐完毕，再看他左、右两边的哲学家，如果他们都处于饥饿且他们左、右两边的哲学家都未进餐时，便可让他们进餐。

(3) test(i:0..4)过程。该过程为测试过程，用它去测试哲学家是否已具备用餐条件，即 state[(k + 4) % 5] ≠eating and state[k] = hungry and state[(k + 1) % 5] ≠eating 条件为真。若为真，方允许该哲学家进餐。该过程将被 pickup 和 putdown 两个过程调用。

用管程解决哲学家进餐问题

例 3-15 用于解决哲学家进餐问题的管程描述如下。

```
Monitor dining－philosophers{
    (thinking, hungry, eating) state[5];
    condition self[5];
    void pickup() (i : 0 . . 4)
    {
       state[ i ] = hungry;
       test( i );
       if (state[ i ] != eating) cwait(self[ i ]);
    }
    void putdown() (i : 0 . . 4)
    {
       state[ i ] = thinking;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }
    void test() (k : 0 . . 4);
    {
       if (state[(k + 4) % 5]!= eating && state[k] == hungry && state[(k + 1) % 5]!= eating)
       {
          state[ k ] = eating;
          csignal(self[ k ]);
       }
    }
    {
       for (i = 0; i<=4; i++)
          state[ i ] = thinking; }
}
```

3.5 进程间消息传递

在计算机系统中，并发进程之间经常要交换一些信息。例如，并发进程间用 PV 操作交换信息实现进程的同步与互斥，保证安全地共享资源和协调地完成任务。因此，PV 操作可看作是进程间的一种通信方式，但这种通信只交换了少量的信号，属于一种低级通信方式。有时进程间要交换大量的信息，这种大量信息的传递要有专门的消息传递机制来实现，这是一种高级的通信方式，也称为“进程通信”。

3.5.1 消息传递的类型

随着操作系统的发展，进程通信机制也在发展，由早期的低级进程通信机制发展为能传送大量数据的高级通信机制。目前，高级通信机制可归结为三大类：共享存储器系统、消息传递系统以及管道通信系统。

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共享存储区，进程之间能够通过它们进行通信。由此，又可把它们进一步分成如下两种类型。

1）基于共享数据结构的通信方式

在这种通信方式中，要求诸进程公用某些数据结构，进程通过它们交换信息。如在生产者-消费者问题中，就是把有界缓冲区这种数据结构用来实现通信。这里，公用数据结构的设置及对进程间同步的处理都是程序员的职责，这无疑增加了程序员的负担，而操作系统却只需提供共享存储器。因此，这种通信方式是低效的，只适于传递少量数据。

2）基于共享存储区的通信方式

为了传输大量数据，在存储器中划出了一块共享存储区，诸进程可通过对共享存储区中的数据进行读或写来实现通信。这种通信方式属于高级通信。进程在通信前向系统申请共享存储区中的一个分区，并指定该分区的关键字；若系统已经给其他进程分配了这样的分区，则将该分区的描述符返回给申请者。接着，申请者把获得的共享存储分区连接到本进程上，此后便可像读/写普通存储器一样地读/写公用存储分区。

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，程序员直接利用系统提供的一组通信命令（原语）来实现通信。操作系统隐藏了通信的实现细节，这大大简化了通信程序编制的复杂性，因而获得广泛的应用，并已成为目前单机系统、多机系统及计算机网络中的主要进程通信方式。消息传递系统的通信方式属于高级通信方式。根据实现方式的不同可分为直接传递方式和间接传递方式。

3. 管道通信系统

所谓管道，是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称为 pipe 文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入管道；而接收管道输出的接收进程（即读进程），可从管道中接收数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于 UNIX 操作系统，因它能传送大量的数据且很有效，故又被引入许多其他操作系统中。

为了协调双方的通信,管道通信机制必须提供以下三方面的协调能力。

(1) 互斥。当一个进程正对 pipe 进行读/写操作时,另一个进程必须等待。

(2) 同步。当写(输入)进程把一定数量数据写满 pipe 时,应睡眠等待,直到读(输出)进程取走数据后,再把它唤醒。当读进程读一空 pipe 时,也应睡眠等待,直至写进程将数据写入管道后,才将它唤醒。

(3) 判断对方是否存在。只有确定对方已存在时,方能进行通信。

3.5.2 直接传递

直接传递是指发送进程利用操作系统所提供的发送命令直接把消息发送给接收进程,而接收进程则利用接收命令直接从发送进程接收消息。在直接通信方式下,企图发送或接收消息的每个进程必须指出信件发给谁或从谁那里接收消息,可用 send 原语和 receive 原语来实现进程之间的通信,这两个原语定义如下。

(1) send(P,消息)。把一个消息发送给进程 P。

(2) receive(Q,消息)。从进程 Q 接收一个消息。

这样,进程 P 和 Q 通过执行这两个操作而自动建立了一种联系,并且这种联系仅仅发生在这一对进程之间。消息可以有固定长度和可变长度两种。固定长度便于物理实现,但使程序设计增加困难;而消息长度可变使程序设计变得简单,但使消息传递机制的实现复杂化。

我们还可以利用直接进程通信原语来解决生产者-消费者问题。当生产者生产出一个消息后,send 原语将消息发送给消费者进程;而消费者进程则利用 receive 原语来得到一个消息。如果消息尚未生产出来,消费者必须等待,直到生产者进程将消息发送过来。

3.5.3 间接传递

采用间接传递方式时,进程间发送或接收消息通过一个共享的数据结构——信箱来进行,消息可以被理解成信件,每个信箱有一个唯一的标识符。当两个以上的进程有一个共享的信箱时,它们就能进行通信。间接通信方式解除了发送进程和接收进程之间的直接联系,在消息的使用上灵活性较大。间接通信方式中“发送”和“接收”原语的形式如下。

(1) send(A,消息)。把一个消息发送给信箱 A。

(2) receive(A,消息)。从信箱 A 接收一个消息。

信箱是存放消息的存储区域,每个信箱可以分成信箱头和信箱体两部分。信箱头指出信箱容量、消息格式、存放消息位置的指针等;信箱体用来存放消息,信箱体分成若干个区,每个区可容纳一个消息。

“发送”和“接收”两条原语的功能如下:

(1) 发送消息。如果指定的信箱未满,则将消息送入信箱中由指针所指示的位置,并释放等待该信箱中消息的等待者;否则,发送消息者被置成等待信箱状态。

(2) 接收消息。如果指定信箱中有消息,则取出一个消息,并释放等待信箱的等待者;否则,接收消息者被置成等待信箱中消息的状态。

两个原语的算法描述如下,其中,W()和 R()是让进程进入等待队列和出队列的两个过程。

```
typedef struct box {
     int size;                                   {信箱大小}
     int count;                                  {现有消息数}
     message letter[n];                          {信箱}
     semaphore s1, s2;                           {等信箱和等消息信号量}
}
void send (box B, message M)
{
     int i;
     if (B.count == B.size) W(B.s1);
     i = B.count + 1;
     B.letter[ i ] = M;
     B.count = i;
     R(B.s2);
}
void receive (box B, message X)
{
   int i;
   if (B.count == 0) W(B.s2);
   B.count = B.count - 1;
   X = B.letter[ 1 ];
   if (B.count≠0)
      for (i = 1; i<=B.count; i++)
         B.letter[ i ] = B.letter[ i + 1 ];
   R(B.s1);
}
```

信箱可由操作系统创建,也可由用户进程创建,创建者是信箱的拥有者。据此,可把信箱分为以下三类。

(1) 私用信箱。用户进程可为自己建立一个新信箱,并作为该进程的一部分。信箱的拥有者有权从信箱中读取信息,其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路信箱实现。当拥有该信箱的进程结束时,信箱也随之消失。

(2) 公用信箱。它由操作系统创建,并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中,也可从信箱中取出发送给自己的消息。显然,公用信箱应采用双向通信链路的信箱来实现。通常,公用信箱在系统运行期间始终存在。

(3) 共享信箱。它由某进程创建,在创建时或创建后指明它是可共享的,同时需指出共享进程(用户)的名字。信箱的拥有者和共享者都有权从信箱中取走发送给自己的消息。

在利用信箱通信时,在发送进程和接收进程之间存在着下述的 4 种关系。

(1) 一对一关系。即可以为发送进程和接收进程建立一条专用的通信链路,使它们之间的交互不受其他进程的影响。

(2) 多对一关系。允许提供服务的进程与多个用户进程之间进行交互,也称为客户/服务器交互。

(3) 一对多关系。允许一个发送进程与多个接收进程进行交互,使发送进程可用广播形式向接收者发送消息。

(4) 多对多关系。允许建立一个公用信箱,让多个进程都能向信箱中投递消息,也可从信箱中取走属于自己的消息。

3.5.4 消息格式

消息的格式取决于消息机制的目标及该机制是运行在一台计算机上还是运行在分布式系统中。对于某些操作系统,设计者优先选用短的、固定长度的消息,以减小处理和存储的开销。如果需要传递大量的数据,数据可以放置到一个文件中,消息可以简单地引用该文件。一种更为灵活的方法是允许可变长度的消息。

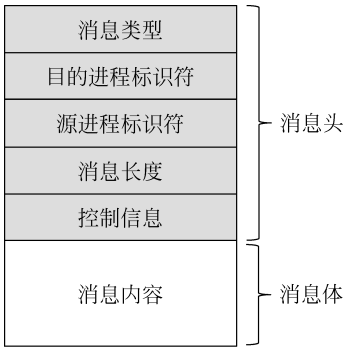

图 3-8 一般消息格式

图 3-8 给出了一种操作系统中支持可变长度消息的典型消息格式。该消息被划分成两部分:包含相关信息的消息头和包含实际内容的消息体。消息头可以包含消息的源和目标的标识符、长度域,以及判定各种消息类型的类型域,还可能含有一些额外的控制信息,例如用于创建消息链表的指针域、记录源和目标之间传递消息的数目、顺序与序号,以及一个优先级域。

3.5.5 解决生产者-消费者问题

作为使用消息传递的一个例子,例 3-16 是解决有界缓冲区生产者-消费者问题的一种方法。该例利用了消息传递的能力,除了传递信号之外还传递数据。它使用了两个信箱。当生产者产生了数据后,它作为消息被发送到信箱 mayConsume,只要该信箱中有一条消息,消费者就可以开始消费。此后 mayConsume 用作缓冲区,缓冲区中的数据被组织成消息队列,缓冲区的大小由全变局变量 capacity 确定。信箱 mayProduce 最初填满了空消息,空消息的数量等于信箱的容量,每次生产使得 mayProduce 中的消息数减少,每次消费使得 mayProduce 中的消息数增长。

这种方法非常灵活,可以有多个生产者和消费者,只要它们都访问这两个信箱即可。系统甚至可以是分布式系统,所有生产者进程和 mayProduce 信箱在一个站点上,所有消费者进程和 mayConsume 信箱在另一个站点上。

例 3-16 使用消息解决有界缓冲区生产者-消费者问题。

```
const int
   capacity = / * 缓冲区容量 * / ;
   null = /* 空消息 * / ;
int i;
void Producer()
{
   message pmsg;
   while (true) {
      receive (mayProduce, pmsg) ;
      pmsg = produce() ;
      send (mayConsume, pmsg) ;
   }
}
void Consumer()
{
   message cmsg;
```

```
    while (true) {
        receive (mayConsume, cmsg);
        consume (cmsg);
        send (mayProduce, null);
    }
}
void main()
{
    create_mailbox (mayProduce);
    create_mailbox (mayConsume);
    for (int i = 1; i <= capacity; i++) send (mayProduce, null);
    parbegin (Producer, Consumer);
}
```

小　　结

现代操作系统的核心是多道程序设计、多处理器和分布式处理器，这些方案的基础及操作系统设计技术的基础是并发。当多个进程并发执行时，无论是在多处理器系统的情况下，还是在单处理器多道程序系统中，都会产生冲突和合作的问题。

由于相关并发进程在执行过程中共享了资源，可能会出现与时间有关的错误，对涉及共享资源的若干并发进程的相关临界区互斥执行，就不会出现与时间有关的错误。可以采用PV操作及管程的方法来解决临界区的互斥问题。

相互合作的一组并发进程，其中每一个进程都以各自独立的、不可预知的速度向前推进，但它们又需要密切合作以实现一个共同的任务，就需要解决进程同步问题。仍可以采用PV操作及管程的方法来解决进程同步问题。各并发进程在执行过程中经常要交换一些信息，我们把通过专门的通信机制实现进程间交换大量信息的通信方式称为“消息传递”，也称为“进程通信”。