第 3 章　同步与通信

3.1　进程的同步与互斥

在进程和线程管理中存在着大量的并发问题，并发执行的进程因为协同实现用户任务或共享计算机资源，使进程之间存在着相互制约的关系。在进程的并发执行过程中存在以下几种制约关系。

（1）间接制约关系。进程彼此之间本来是无关的，但是由于竞争使用同一共享资源而产生相互制约的关系，这种因共享资源而产生的制约关系是一种间接制约关系，即进程的互斥。例如，有进程 A 和进程 B，如果在进程 A 提出打印请求时，系统已经将唯一的一台打印机分配给了进程 B，那么此时进程 A 只能被阻塞；而只有当进程 B 将打印机释放后，进程 A 才会由阻塞态变为就绪态。

（2）直接制约关系。进程之间通过在执行时序上的某种限制而达到相互合作的约束关系是一种直接制约关系，即进程的同步。进程之间相互清楚对方的存在及作用，相互交换信息，往往指有几个进程共同完成一个任务。例如，有一个输入进程 A，通过单缓冲向计算进程 B 提供数据。当缓冲区为空时，进程 B 因不能获得所需要的数据而产生阻塞，而当进程 A 把数据输入缓冲区后，便将进程 B 唤醒；反之，当缓冲区已满时，进程 A 因不能再向缓冲区放数据而产生阻塞，当进程 B 将缓冲区中的数据取走后便可以唤醒进程 A。

3.1.1　基本概念

1．临界资源

我们在第 1 章中介绍过，凡是以互斥方式使用的共享资源都称为临界资源（Critical Resource）。临界资源具有一次只允许一个进程使用的属性。例如，计算机的许多硬件资源都被处理成临界资源，如打印机、磁带机等。如果打印机允许若干用户同时打印，那么打印结果会混淆在一起，不易分辨，给用户带来许多不便。系统中还有许多软资源，如内存中的公共数据结构、共享变量、表格、队列、栈等也被处理成临界资源，以避免多个进程对其访问时出现问题。

2．临界区

在对临界资源的访问过程中，进程自身需要以某种方式表达互斥的要求。例如，在访问前将资源锁定，访问过程中不被打扰，访问完成后开锁，之后才允许其他进程使用。每个进程中，访问临界资源的那段代码称为临界区（Critical Section，CS）。如果能保证各个

进程互斥地进入自己的临界区，便可实现各个进程对临界资源的互斥访问。把申请进入临界区的那段代码称为进入区，如果可以进入，那么设置“正在访问”标志。相应地，在临界区之后，用于退出对临界区访问的那段代码称为退出区，此时将临界区的标志恢复为“未被访问”。进程中，除进入区、临界区及退出区之外的代码称为剩余区。使用临界资源的程序结构如图 3-1 所示。

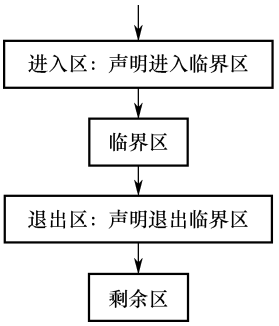

图 3-1　使用临界资源的程序结构

3．互斥准则

进程必须要互斥地进入自己的临界区，所以操作系统无论提供何种同步机制，都要满足互斥的基本要求。总之，所有的同步机制都应遵循以下 4 条互斥准则。

（1）空闲让进。当没有进程在临界区时，任何需要进入临界区的进程都允许立即进入。

（2）忙则等待。在共享同一对象的所有进程中，一次只能有一个进程进入临界区，其他要求进入临界区的进程只能等待。

（3）有限等待。任何一个进程经有限时间等待后都能进入临界区，以免陷入“死等”状态。

（4）让权等待。当一个进程不能进入临界区时要立即阻塞自己，释放处理机让其他进程使用，避免“忙等”。

需要说明的是，（4）是为了提高系统效率而提出的。因为如果允许“忙等”，那么就意味着进程在请求进入临界区的同时，可能要消耗大量的处理机时间。

3.1.2　硬件同步机制

为解决进程互斥进入临界区的问题，可以为每类临界资源设置一把锁，该锁有打开和关闭两种状态。进程必须在访问前将资源锁定，以确保访问过程中不被打扰，访问完成后开锁，之后才允许其他进程使用。而在硬件上，由于对一个存储单元的访问是互斥进行的，所以可以利用处理机提供的特殊指令实现对临界区加锁。在设计中，可扩展指令功能，让一条指令实现两个操作，以确保其原子性。最常见的硬件指令有测试与设置指令和交换指令。

1．测试与设置指令

测试与设置指令（Test_and_Set）可定义如下：

电子工业出版社版权所有 盗版必究

```
boolean Test_and_Set(int i)
       {  if(i==0)
                {  i=1;
                   return true;
                }
          else {
                return false;
               }
        }
```

该指令用于测试参数 i 的值。若参数 i 值为 0，则用 1 取代它并返回 true；否则，参数 i 的值不变并返回 false。整个测试与设置指令的执行是一个原子操作，即它的执行过程不会被中断。

设 lock 为全局布尔变量，初始值为 0，表示没有进程进入。利用测试与设置指令，即可实现对临界区的加锁与解锁。

```
do {  while (!Test_and_Set(lock)) do skip;  //循环等待，entry section
     critical section;
     lock=false;     //exit section
     remainder section;
}while(true);
```


2．交换指令

交换指令可将两个指定位置上的数据进行交换。交换指令（Exchange）可定义如下：

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

设 lock 为全局布尔变量，初值为 0，表示没有进程进入，每个进程设置一个局部布尔变量 key。利用交换指令可实现对临界区的加锁与解锁。

```
do{  int key=1;
      while(true){
                while (key==1) Exchange(key,lock);
                critical section;
                Exchange(key,lock)
                remainer section;
               }
      }while(true);
```

3．使用硬件指令方法的特点

使用硬件指令实现互斥有以下优点。

（1）适用于单处理机或共享内存的多处理机上的任何数目的进程。

电子工业出版社版权所有 盗版必究

（2）非常简单且易于证明。

（3）可用于支持多临界区，每个临界区可以用其自己的变量定义。

使用硬件指令实现互斥也存在一些严重的缺点。

（1）使用了忙等待。因此，当一个进程正在等待进入一个临界区时，它会继续消耗处理机时间。

（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，某些进程可能无限期地被拒绝进入，这些进程此时就处于饥饿状态，即指进程长时间得不到处理机而无法执行。

（3）可能死锁。例如，在单处理机系统中，进程 P1 执行专门指令（如测试与设置指令、交换指令等）并进入临界区，然后进程 P1 被中断并把处理机让给具有更高优先级的进程 P2。如果进程 P2 试图使用与进程 P1 相同的资源，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，由于进程 P1 的优先级比进程 P2 低，它将永远不会被调度执行。

综上所述，利用这些硬件指令虽然可以实现临界区的互斥访问，但是仍然存在缺陷。因此，需要寻找其他机制来解决问题。

3.1.3 信号量机制

著名的荷兰计算机学者 Dijkstra 经过长期对操作系统研究与设计，于 1965 年提出了一种同步机制——信号量机制。信号量机制由一个称为信号量（Semaphore）的特殊变量和两个被命名为 P 操作和 V 操作的原语（分别对应 wait( )和 signal( )）组成。在这里，原语是指完成某种功能且不被分割、不被中断执行的操作序列。

信号量机制在广泛的应用中得到了迅速发展，它是一种比硬件指令更加完善的同步机制。其基本原理是，为每个临界资源设置一个信号量，负责在多个进程之间转发互斥信息。当一个进程需要互斥使用某个临界资源时，可以通过执行对信号量的 P 操作，了解该资源的空闲情况。当它使用完该临界资源后，又可以通过执行对相关信号量的 V 操作，让其他需要使用该资源的进程感知到该临界资源已经可以使用。

若一个资源被视为临界资源，则应当为它定义一个独立的信号量，对进程访问起“放行”和“阻止”作用，就像交通管理中每个路口设置的信号灯一样。

1．整型信号量

最初由 Dijkstra 把信号量定义为一个整型信号量，代表资源数目或可同时使用该资源的进程个数。信号量的初值为非负数，除初始化外，仅能通过两个标准的原子操作 wait(S)和 signal(S)来访问和修改。

wait(S)和 signal(S)操作可描述为：

```
semaphore  S;
wait(S):   while S<=0 do no-op
           S=S-1;
signal(S): S=S+1;
```

电子工业出版社版权所有

在信号量机制的 wait(S)操作中，只要信号量 S≤0，就会不断地测试，直到 S>0 才会停止。因此，该机制并未遵循“让权等待”的准则，而是使进程一直处于“忙等”状态。

2．记录型信号量

记录型信号量是一种不存在“忙等”现象的进程同步机制。但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除需要一个用于代表资源数目的整型变量 count 外，还应增加一个进程链表指针 queue，用于链接上述所有等待进程。此信号量定义如下：

```
typedef struct {
               int  count;
               queueType *queue;
              } semaphore;
```

在该定义中，count 是信号量值，其值为正时，表示某种资源的数量；queue 表示信号量等待队列指针，当信号量值为负时，表示该类资源已分配完毕，等待该类资源的进程只能排在等待队列中。

相应地，wait(S)操作可描述为：

```
semaphore  S;
void wait(S)
{
    S.count - -;
    if (S.count<0)
    {
        block(S.queue);    //阻塞该进程
        链接到 S.queue 队列;
      }
}
```

wait(S)操作用于进入临界区前进行资源申请。S.count 的初值表示系统中某类资源的数目，因此该信号量又称为资源信号量，对它的每次 wait(S)操作，意味着进程请求一个单位的该类资源，因此描述为 S.count−−。当 S.count＜0 时，表示该类资源已分配完毕，因此进程应调用 block( )原语，进行自我阻塞，放弃处理机，并插入到信号量链表 S.queue 中。可见，该机制遵循了“让权等待”准则。此时 S.count 的绝对值表示在该信号量链表中因申请资源不满足而阻塞的进程数目。

signal(S)操作可描述为：

```
void signal(S)
  {  S.count ++;
     if (S.count＜=0) wakeup(S.queue);
           //从 S.queue 队列中移出一个进程，把该进程链接到就绪队列
  }
```

signal(S)用于退出临界区后，释放资源。对信号量的每次 signal( )操作，表示执行进程释放一个单位资源，故 S.count++操作表示资源数目加 1。若资源数目加 1 后仍有 S.count≤0，则表

电子工业出版社版权所有 盗版必究

示在该信号量链表中，仍有等待该资源的进程被阻塞，故应调用 wakeup( )原语，将 S.queue 链表中的第一个等待进程唤醒。若 S.count 的初值为 1，则表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

3．信号量的使用

（1）用于实现互斥。例如，用于 *n* 个进程的临界区之间互斥，设 *n* 个进程共享一个信号量 mutex，初值为 1。为了实现进程互斥地进入临界区，只需把临界区置于 wait(mutex)和 signal(mutex)之间。任一进程 Pi 的框架为：

```
void Pi ( )
{   semaphore mutex=1; // 信号量初值为 1
    do{
          wait(mutex);
          critical section;
          signal(mutex);
          remainder section;
      }while (true);
}
```


（2）用于实现同步。例如，有两个并发进程 P1 和 P2，共享一个公共信号量 S，初值为 0。P1 执行的程序中有一条 S1 语句，P2 执行的程序中有一条 S2 语句。而且，只有当 P1 执行完 S1 语句后，P2 才能开始执行 S2 语句。对于这种同步问题，可以很容易地用信号量机制解决。进程间的同步控制描述如下：

```
semaphore S=0;  // 信号量初值为 0
//进程 P1 程序框架如下:
void P1( ){
              …
              S1;
              signal(S);
              …
          }
//进程 P2 程序框架如下:
void P2( ){
              …
              wait(S);
              S2;
              …
          }
void main( )
       {
          Parbegin(P1( ),P2( ));
       }
```

电子工业出版社版权所有 盗版必究

在这里，Parbegin 的作用是挂起主程序，初始化并发进程 P1 和 P2。

3.2 经典进程同步问题

3.2.1 生产者-消费者问题

生产者-消费者（Producer-Consumer）问题是一个著名的进程同步问题。它描述的是：有若干生产者进程在生产产品，假设生产者进程为 k（k>0）个，并将这些产品提供给 m（m>0）个消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有 n 个缓冲区的缓冲池（假设组织成环形），生产者进程将所生产的产品放入缓冲区中；消费者进程可从一个缓冲区中取出产品。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步。假定它们的约束条件有以下 4 个。

（1）当缓冲池中有空缓冲区时，允许任一生产者进程把产品放入。

（2）当缓冲池中无空缓冲区时，则试图将产品放入缓冲区的任何生产者进程必须等待。

（3）当缓冲池中有产品时，允许任一个消费者进程把其中的一个产品取出消费。

（4）当缓冲池中没有产品时，试图从缓冲池内取出产品的任何消费者进程必须等待。

对所有生产者和消费者进程来说，可以把缓冲池视为一个整体，缓冲池是临界资源，即任何一个进程在对缓冲池中某个缓冲区进行“放入”放“取出”操作时必须和其他进程互斥执行。下面用信号量机制解决这个问题，首先定义信号量如下：

（1）信号量 mutex，初值为 1，用于控制互斥地访问缓冲池。

（2）信号量 full，初值为 0，用于资源计数。full 值表示当前缓冲池中“满”缓冲区的个数。

（3）信号量 empty，初值为 n，用于资源计数。empty 值表示当前缓冲池中“空”缓冲区的个数。

生产者-消费者问题解决方案如下：

```
/*program producerconsumer*/
const int n;    // buffer size
int in, out=0; //缓冲区首空、首满指针
semaphore empty=n;
semaphore full=0;
semaphore mutex=1;
void producer( )
{
   while(true)
    {
       生产一个产品;
        wait(empty);
        wait(mutex);
      将产品放入缓冲区;
```


电子工业出版社版权所有 盗版必究

```
        in=(in+1) % n;
        signal(mutex);
        signal(full);
    }
  }
  void consumer( )
  {
    while(ture)
      {   wait(full);
          wait(mutex);
       从缓冲区中取产品;
          out=(out+1) % n;
          signal(mutex);
          signal(empty);
          消费该产品;
      }
  }
  void main( )
  {   parbegin( producer( ),consumer( ) ); }
```


在这个问题中，生产者进程和消费者进程分别有两个 wait 操作和两个 signal 操作。

思考：如果将两个 wait 操作互换或将两个 signal 操作互换，结果会如何？

我们以消费者进程互换 wait 操作为例进行说明。假设消费者进程的两个 wait 操作互换，那么考虑一种供不应求的情况，即消费者进程消费得快，生产者进程生产得慢，则总有一个时刻会出现缓冲区全空，消费者进程先执行的情况。这时消费者进程如果先执行互斥信号量的 wait 操作，则会锁定缓冲区，然后再申请产品资源，那么会因为缓冲区中没有产品而阻塞在第二个 wait 操作处，直到缓冲区中有产品时才能被唤醒。而此时由于缓冲区已被锁定，即使生产者进程有生产能力，也无法进入缓冲区放入产品，所以生产者进程会在第一个 wait 操作处阻塞，直到消费者进程释放缓冲区。这样，就使得生产者进程和消费者进程因互相等待而均处于阻塞态，系统进入一种僵死状态，即死锁（我们将在本书第 4 章中详细阐述）。

特别注意：进程中若有多个 wait 操作，同步信号量的 wait 操作应该在前，而互斥信号量的 wait 操作应该在后，以免引起死锁。进程中若有多个 signal 操作，其操作顺序无所谓。

3.2.2 读者-写者问题

读者-写者问题描述如下：有一个数据区（数据区可以是一个文件、一块内存空间或一组寄存器）被多个用户共享，其中一部分用户是读者，另一部分是写者。我们规定：读者对数据区是只读的，而且允许多个读者同时读；写者对数据区是只写的，当一个写者正在向数据区写信息的时候，不允许其他用户使用。即保证一个写者进程必须与其他进程互斥地访问共享对象。

西安工业出版社版权所有 盗版必究

解决该问题必须满足的条件如下：

（1）任意多个读者可以同时读；

（2）任一时刻只能有一个写者可以写；

（3）如果写者正在写，那么读者就不能读。

该问题中的数据区作为多个用户共享的资源，只要有读者使用时就不允许任何一个写者使用。当一个写者使用时，不允许其他写者或读者使用。因此，可以将数据区访问的程序视为临界区。但是，数据区不是绝对的临界资源，因为它允许多个读者用户同时使用。

下面我们用两种方法来解决该问题。一种是读者优先，另一种是写者优先。读者优先是指一旦有一个写者访问数据区，只要还有一个读者在进行读操作，后续的读者就不需要等待，可以保持对数据区的控制，而写者必须等待所有的读者读完才可以进行写操作。

首先，使用一个互斥信号量 wmutex，实现读者与写者、写者与写者之间的互斥。由于读者可以同时访问数据区，因此我们让第一个准备进入临界区的读者，通过 wait(wmutex) 强占临界资源，以便排斥写者访问。当最后一个读者离开临界区时，通过 signal(wmutex) 将临界资源释放，以便其他用户使用。

其次，设计一个专为读者共享的变量 readcounter，用来记录读数据区的人数。readcounter 应视为临界资源，故应有一个互斥信号量，定义为 rmutex。

读者优先的方法描述如下：

```
/*program readersandwriters*/
int readcounter;
semaphore wmutex=1;
semaphore rmutex=1;   //对 readcounter 的互斥访问
void reader( )
{
    while(true)
    {
        wait(rmutex);
        readcounter++;
        if (readcounter==1) wait(wmutex);
        signal(rmutex);
        readunit( );
        wait(rmutex);
        readcounter--;
        if (readcount==0) signal(wmutex);
        signal(rmutex);
    }
}
void writer( )
{
    while(true)
    {
```

电子工业出版社版权所有
盗版必究

```
            wait(wmutex);
            writeunit( );
            signal(wmutex);
        }
    }
    void main( )
    {
         readcounter=0;
         parbegin(reader( ),writer( ));
    }
```

在该方法中规定读者优先，写者即使早于读者来，只要写者来之前有读者在读，写者就必须等待，等待读者读完离开后，写者再对数据区进行控制，这容易导致写者饥饿。下面我们分析写者优先的解决方法，只要有写者申请写操作，就不允许新的读者访问数据区。

在前面定义的基础上，写者优先的解决方法增加了以下的信号量和变量：

（1）信号量 rsem 用于在有写者访问数据区时，屏蔽所有的读者；

（2）变量 writecount 控制 rsem 的设置；

（3）信号量 y 控制 writecount 的修改。

除此之外，对于读进程，还必须添加一个额外的信号量 z。因为在 rsem 上不允许多于一个进程在排队，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 rsem 上排队，而所有其他读进程在等待 rsem 之前，在信号量 z 上排队。表 3-1 总结了写者优先方法中进程队列的各种情况。

表 3-1　写者优先方法中进程队列的各种情况

队 列 状 态	操　　作
系统中只有读者	设置信号量 wsem；没有排队
系统中只有写者	设置信号量 wsem 和 rsem；写者在信号量 wsem 上排队
存在读者和写者且读者优先	读者设置信号量 wsem；写者设置信号量 rsem；所有的写者在 wsem 上排队；只有一个读者在 rsem 上排队，其他读者在信号量 z 上排队
存在读者和写者且写者优先	写者设置信号量 wsem；写者设置信号量 rsem；写者在 wsem 上排队；只有一个读者在 rsem 上排队，其他读者在信号量 y 上排队

写者优先的方法描述如下：

```
/*program readersandwriters */
int readcount,writecount;
semaphore x=1,y=1,z=1,wsem=1,rsem=1;
void reader( )
{
  while(true)
   {
      wait(z);
      wait(rsem);
```

电子工业出版社版权所有 盗版必究

```
        wait(x);
             readcount++;
        if (readcount==1) wait(wsem);
        signal(x);
        signal(rsem);
        signal(z);
        readunit( );
        wait(x);
        readcount--;
        if (readcount==0) signal(wsem);
        signal(x);
       }
}
void writer( )
{
   while(true)
     {
        wait(y);
        writecount++;
        if (writecount==1) wait(rsem);
        signal(y);
        wait(wsem);
        writeunit( );
        signal(wsem);
        wait(y);
        writecount--;
        if (writecount==0) signal(rsem);
        signal(y);
     }
}
void main( )
{    readcount=writecount=0;
     parbegin(reader( ),writer( ));
}
```

电子工业出版社版权所有
盗版必究

3.2.3 哲学家就餐问题

Dijkstra 于 1965 年首先提出并解决了哲学家就餐问题，该问题也是大量并发控制问题中的一个典型例子。

哲学家就餐问题描述如下：有 5 位哲学家，他们倾注毕生精力用于思考问题和吃饭。设想他们共同坐在一张放有 5 把椅子的餐桌前用餐，用餐过后开始思考问题。他们的生活方式可描述为一个单调的循环：思考-饥饿-用餐-思考。已知餐桌上摆的是面条，桌上放着

五根筷子，任意两个哲学家之间有一根筷子，如图 3-2 所示。哲学家在思考问题时，并不影响他人。只有当哲学家饥饿的时候，他才试图拿起左边和右边的筷子（一根一根地拿起）。若筷子已在他人手中，则需等待。饥饿的哲学家只有左右手各拿到一根筷子才可以开始进餐。而且，也只有当他吃完饭后才放下筷子，重新开始思考问题。

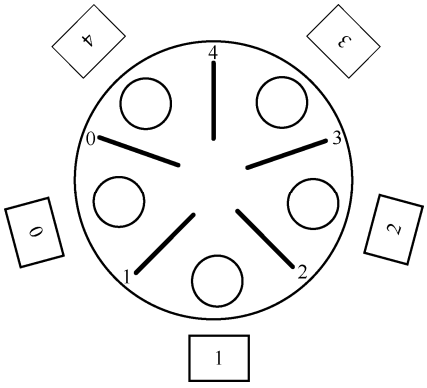

图 3-2　哲学家就餐问题

在哲学家就餐问题中，筷子是一种临界资源。设 *i* 号筷子的互斥信号量为 chopstick[*i*](*i*=0,1,2,3,4)，初值均为 1。哲学家就餐问题解决方案如下：

```
/*program diningphilosophers*/
semaphore chopstick[5]={1,1,1,1,1];
int i;
void Philosophers (int i)
{
    while(true)
    {
        think( );
        wait(chopstick[i] );
        wait(chopstick[(i+1) mod 5] );
        eat( );
        signal(chopstick[i]);
        signal(chopstick[(i+1) mod 5] );
    }
}
void main( )
{
 parbegin(Philosophers(0),Philosophers(1),Philosopher(2),Philosopher(3),
 Philosopher(4) );
}
```

显然，上述解决方案可以保证不会有两位相邻的哲学家同时进餐，但有可能引起死锁。假设 5 位哲学家同时饥饿而各自拿起左边的筷子时，就会使 5 个信号量 chopstick 均为 0；当他们再试图去拿右边的筷子时，都将因无筷子可拿而无限期地等待。对于这样的死锁问

电子工业出版社版权所有
盗版必究

题，可采取如下两种解决方法。

（1）规定奇数号哲学家先拿左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，0、1 号哲学家竞争 1 号筷子；2、3 号哲学家竞争 3 号筷子。

（2）至多有 4 位哲学家同时拿筷子。

解决方法（1）的哲学家活动可描述为：

```
void Philosophers(int i)
{
    while(true)
       {
          think( );
          if( i%2==0)
             {
                wait(chopstick[i] );
                wait(chopstick[(i+1) mod 5]);
             }
          else
             {
                wait(chopstick[(i+1) mod 5]);
                wait(chopstick[i]);
             }
          eat( );
          signal(chopstick[i]);
          signal(chopstick[(i+1) mod 5]);
       }
}
```

解决方法（2）可描述为：

```
/*program diningphilosophers*/
 semaphore chopstick[5]={1,1,1,1,1];
int i;
semaphore room=4;  // 用于控制同时进餐的哲学家位数
void Philosophers (int i)
{
   while(true)
      {   think( );
          wait(room);
          wait(chopstick[i]);
          wait(chopstick[(i+1) mod 5] );
          eat( );
          signal(chopstick[i]);
          signal(chopstick[(i+1) mod 5] );
          signal(room);
```


电子工业出版社版权所有
盗版必究

```
        }
    }
    void main( )
    {
    parbegin( Philosophers(0), Philosophers(1), Philosopher(2),
    Philosopher(3), Philosopher(4) );
    }
```

以上所介绍的生产者-消费者问题，读者-写者问题及哲学家就餐问题是进程同步和互斥的典型实例，我们可以总结出这类问题的解题思路。

解题思路：首先要分清哪些是互斥问题，哪些是同步问题。对于那些访问临界资源的问题一般属于互斥问题，而具有前后执行顺序要求的属于同步问题。当然，有的问题可能既涉及同步又涉及互斥。然后，对于互斥问题，应设置互斥信号量，不管有互斥关系的进程有几个或几类，通常只设置一个互斥信号量，且初值为 1，代表一次只允许一个进程对临界资源访问。而对于同步问题，应设置同步信号量，通常同步信号量的个数与参与同步的进程种类有关，即同步关系涉及几类进程，就有几个同步信号量。同步信号量表示该进程是否可以开始或该进程是否已经结束。

3.3 管程

信号量机制为实现互斥和进程间的协调提供了一个功能强大而灵活的工具，然而，使用信号量来编写正确的程序是比较困难的。由此，便产生了一种新的进程同步工具——管程。

3.3.1 管程的基本概念

1. 管程的引入

信号量机制是一种既方便又有效的进程同步机制，但采用信号量机制编写并发程序时，对临界区的执行会分散在各个进程中，其缺点如下。

（1）易读性差。要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序。

（2）不利于修改和维护。因为程序的局部性很差，所以任一组变量或一段代码的修改都可能影响大局。

（3）正确性难以保证。因为操作系统或并发程序通常很大，要保证这样一个复杂的系统没有逻辑错误是很难的。

这不仅给系统的管理带来了麻烦，而且会因同步操作的使用不当而导致系统死锁。为了解决这一问题，著名学者 Hoare 提出了一种具有面向对象程序设计思想的同步机制——管程。它提供了与信号量机制相同的功能，但是更易于控制，因而在很多系统中得到实现，如 Pascal、Java、Modula-3 等。

电子工业出版社版权所有 盗版必究

2．管程的概念

管程是一种并发性的结构，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。管程主要由 4 部分组成。

（1）管程名；

（2）局部于管程的共享变量说明；

（3）对该数据结构进行操作的一组过程或函数；

（4）对局部于管程的数据设置初始值的语句。

管程的示意图如图 3-3 所示。

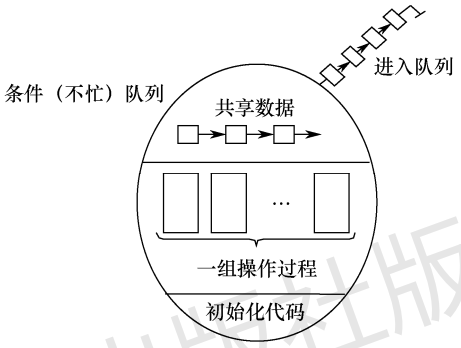

图 3-3　管程的示意图

管程的语言结构可描述为下述形式：

```
monitor monitor-name;    //定义管程名
variable declarations;  //局部于管程的共享变量声明
initialization code; //初始化代码
P1(…){…};//一组操作
P2(…){…};
…
Pn(…){…};
```

管程最主要的特点如下：

（1）只能通过管程中的过程而不能用其他外部过程访问其局部数据变量；

（2）进程通过调用管程的过程而进入管程；

（3）每一时刻只能有一个进程在管程中执行，任何其他调用管程的进程将被挂起直至管程可用为止。

3．条件变量

管程在任一时刻只允许一个进程调用，这样就可以实现互斥，任一时刻管程中的局部数据也只能被一个进程访问。因此，一个共享的数据结构可以放在管程中加以保护，如果管程中的数据代表某种资源，那么管程就支持对这种资源访问的互斥。

为了将管程用于并发进程中，管程必须拥有同步手段。例如，假设一个进程调用了管程，当它在执行过程中发现条件不满足时，应将该进程阻塞。这就需要一种机制，使得该

电子工业出版社版权所有 盗版必究

进程不仅被阻塞，而且能释放这个管程，以便其他进程可以进入。以后，当条件满足并且管程再次可用时，需要恢复该进程并允许它在中断点重新进入管程。为了区分阻塞的原因，引入了局部于管程的条件变量，即 condition c。

管程可使用条件变量支持同步，这些变量保存在管程中并且只能在管程内部访问。用以下两个原语操作条件变量。

c.wait( )：用来将执行进程排在与条件变量 c 相应的等待队列上。

c.signal( )：用来唤醒与 c 相应的等待队列上的一个进程。若没有等待进程，则 c.signal( ) 不起任何作用，什么都不做。

需要注意的是，管程中的 c.wait( )操作和 c.signal( )操作与信号量机制中的 wait 操作和 signal 操作是不同的，具体表现在：

（1）使用 c.wait( )操作会挂起当前进程，而执行 wait 操作未必产生阻塞；

（2）如果没有等待进程，c.signal( )操作什么都不做，这一点与 signal 操作不同。

3.3.2 用管程解决生产者-消费者问题

在利用管程来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为 Producer-Consumer，简称为 PC。

（1）管程中有两个条件变量：

notfull：缓冲区未全满，即缓冲区中还有空位置时，变量值为 true；

notempty：缓冲区未全空，即缓冲区中至少有一个产品存在，变量值为 true。

（2）为该管程定义两个函数（假设缓冲区中的产品是字符）：

放产品：append(char x)；

取产品：take(char x)。

使用管程解决生产者-消费者问题的方法如下：

```
/* program producer-consumer */
monitor PC;      //管程名
char buffer[N]; int nextin, nextout; int count;   //局部变量
condtion notfull, notempty;  //条件变量
void append(char x)
{
   if (count==N) notfull.wait( );
   buffer[nextin]=x;
   nextin=(nextin+1)%N;
   count++;
   notempty.signal( );
 }
void take(char x)
{
   if (count=0) notempty.wait( );
   x=buffer[nextout];
```

电子工业出版社版权所有 盗版必究

```
    nextout=(nextout+1)%N;
    count--;
    notfull.signal( );
  }
{  nextin=0; nextout=0; count=0;   //局部变量初始化
  }
void producer( )
{
  char x;
  while(true)
   {
     produce(x);
     append(x);
    }
}
void consumer( )
{
   char x;
   while(true)
     {
        take(x);
        consume(x);
     }
}
void main( )
{
   parbegin(producer(), consumer());
}
```

电子工业出版社版权所有
盗版必究

由此看出，与信号量机制相比，管程担负的责任不同。对于管程，它构造了自己的互斥机制，即生产者和消费者不可能同时访问缓冲区；但是程序必须把适当的 c.wait( )和 c.signal( )原语放在管程中，用于防止进程往一个满缓冲区中放入产品，或者从一个空缓冲区中取出产品。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。除此之外，与信号量机制相比，管程的优势在于所有的同步机制都限制在管程内部，因此不仅易于验证同步的正确性，而且易于检测错误。

3.4 进程通信

在多道程序环境中，许多进程需要互相交换信息，其所交换的信息量，少则仅有几个字节，甚至只有一个状态标志，多则有成千上万个字节。通常，将进程间的这种数据交换称为进程通信。

一般来说，进程间的通信根据通信内容可以划分为两种：即控制信息的传送与大批量数据传送。有时，我们也把进程间控制信息的传送称为低级通信，而把进程间大批量数据传送称为高级通信。上面几节中所介绍的进程间同步或互斥是通过使用信号量进行通信来实现的，属于低级通信。低级通信一般只传送一个或几个字节的信息，以达到控制进程执行速度的目的，高级通信则要传送大量数据。高级通信不是为了控制进程的执行速度，而是为了交换信息。通常，高级通信机制可归结为三大类：共享存储器系统、管道通信系统及消息传递系统。

3.4.1 共享存储器系统

共享存储器系统（Shared-Memory System）需要在内存中开辟一个共享存储区，供进程之间交换信息。两个需要互相交换信息的进程通过对同一共享数据区（Shared Memory）的操作来达到互相通信的目的，而这个共享数据区是每个互相通信进程的一个组成部分。计算进程将所得的结果送入内存共享区的缓冲区中，打印进程从中将结果取出来，就是一个利用共享存储器系统进行通信的例子。

在共享存储器系统中，共享存储区一般应当是需要互斥访问的临界资源。诸多进程为了避免丢失数据或重复取数，需要执行特定的同步协议。

在利用共享存储器系统进行通信之前，信息的发送者和接收者都要将共享存储区纳入自己的虚地址空间中，让它们都能访问该区域。存储器管理模块将共享空间映射成实际的内存。

如图 3-4 所示，有进程 A 和进程 B，进程 A 的虚地址空间 A-2 和进程 B 的虚地址空间 B-2 被映射到内存的共享存储区中，它们可以通过共享存储区实现通信。若将共享存储区又映射到另外的进程空间中，则这个进程也可以参与到进程 A 和进程 B 的通信中。

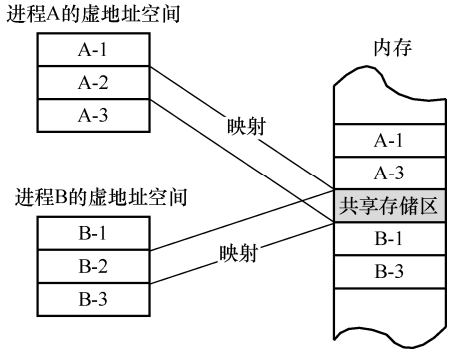

图 3-4　地址空间映射示意图

共享存储器系统具有以下特点：

（1）不要求数据移动；

（2）通信的效率特别高，适用于通信速度要求特别高的场合；

（3）这种同步与互斥机制的实现一般要由程序员来承担，系统仅提供一个共享内存空间的管理机制。

电子工业出版社版权所有 盗版必究

3.4.2 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数据通信的一个共享文件，又称为 pipe 文件。由于写进程和读进程是利用管道进行通信的，所以这种通信方式被称为管道通信。它最早应用于 UNIX 中，由于它能有效地传送大量数据，所以后来被引入到许多其他操作系统中。

在管道通信系统中，向管道提供输入的发送进程（写进程），以字符流的形式将大量的数据写入管道；而接收管道输出的接收进程（读进程），则从管道中读取数据。

最简单的一种管道是批处理命令中使用管道符号“|”建立的。例如：

```
who|sort
```

who 的输出数据经管道传递给 sort 作为输入。这样，一个管道连接了两个命令，形成了一个管道行，数据流从左向右单向流动。

为了协调管道通信双方，管道通信系统必须提供 3 个方面的协调能力。

（1）互斥问题。对管道文件的访问应当互斥地进行，当一个进程对管道进行读写时，另一个进程不可以访问它。

（2）同步问题。当写进程把一定数量的数据写入管道时，会将自己阻塞，直到读进程取走数据后，再把它唤醒。当读进程读一个空管道时，由于此时没有数据可取，也会进入阻塞，直到写进程将数据写入管道后，再将它唤醒。

（3）状态测试。读进程和写进程都能以一种方式了解通信对方是否存在。只有确定对方已经存在时，才能进行通信。

3.4.3 消息传递系统

不论是单处理机系统、多处理机系统，还是计算机网络，消息传递都是用得最广泛的一种进程间通信机制。在消息传递系统中，进程间的数据交换是以格式化的消息（message）为单位的。在计算机网络中，消息又称为报文。

消息传递中的管理机制由操作系统提供，程序员可以直接利用系统提供的一组通信原语进行通信，主要有以下两条原语。

发送原语：send(destination, message)，其中，destination 为消息的目的地（接收进程名）。该原语表示发送消息到进程 destination。

接收原语：receive(source, message)，其中，source 是消息发出地（发送进程名）。该原语表示从进程 source 接收消息。

由于该方式隐藏了通信的实现细节，大大降低了通信程序编写的复杂性，适应性很强，所以获得了广泛应用，能在分布式系统、共享存储器的多处理机系统以及单处理机系统等不同系统中实现。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

1. 直接通信方式

直接通信方式是指发送进程利用操作系统所提供的发送原语，直接把消息发送给接收进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，操作系统提供下述两条通信原语：

电子工业出版社版权所有
盗版必究

```
send(receiver, message);   //发送一个消息给接收进程
seceive(sender, message);  //接收发送进程发来的消息
```

例如，原语 send(P2, m1)表示将消息 m1 发送给接收进程 P2；而原语 receive(P1, m1)则表示接收由发送进程 P1 发来的消息 m1。

在某些情况下，接收进程可与多个发送进程进行通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。对于此类应用，使用隐式的寻址更为有效。因此，在接收进程接收消息的原语中，source 为接收操作执行后的返回值。receive 原语可表示为：

```
receive (source, message);
```

根据上面的讨论，我们可以利用直接通信原语来解决生产者-消费者问题。当生产者进程生产出一个产品（消息）后，便用 send 原语将产品（消息）发送给消费者进程；而消费者进程则利用 receive 原语来接收产品（消息）。如果产品（消息）尚未生产出来，消费者进程必须等待，直至生产者进程将产品（消息）发送过来。生产者-消费者的通信过程可分别描述如下：


```
void producer( )
{  do{    …
        produce an item in nextp
        …
        send(consumer, nextp);
    }while (true);
 }
void consumer( )
{
    do{  …
       receive(producer, nextc);
      …
       consume the item in nextc;
    }while (true);
}
```

2．间接通信方式

间接通信方式是指进程间的通信需要通过共享的数据结构作为中间实体来暂存发送进程发送给接收进程的消息。该中间实体称为信箱，接收进程从信箱中取出发送进程发送给自己的消息。消息可以在信箱中安全地保存，所以利用信箱既可以实现实时通信，也可以实现非实时通信。

系统提供了若干原语用于信箱通信。

（1）信箱的创建和撤销。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性（公用、私用或共享）等；对于共享信箱，还应给出共享者的名字。当进程不再需要信箱时，可用信箱撤销原语将之撤销。

（2）消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并

电子工业出版社版权所有 盗版必究

利用系统提供的通信原语进行通信。

```
send(mailbox, message):   //将一个消息发送到指定信箱
receive(mailbox, message): //从指定信箱中接收一个消息
```

在这里需要指出的是，信箱按其属性可分为三类.

（1）私用信箱。私用信箱是指由用户进程建立并作为该进程其中一部分的信箱。它可以采用单向通信链路的信箱来实现。信箱的拥有者有权从信箱中读取消息，其他用户只能将消息发送到该信箱中。当拥有该信箱的进程结束时，信箱也随之消失。

（2）公用信箱。公用信箱是指由操作系统创建，并提供给系统中的所有核准进程使用的信箱。它可以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。共享信箱由某进程创建，并在创建时指明它可以被哪些用户进程共享。信箱的拥有者和共享者都有权访问信箱，从中取走发送给自己的消息。

我们也可以通过两个信箱 producemail 和 consumemail 来解决生产者-消费者问题，信箱中的消息组织成消息队列，信箱的容量由全局变量 capacity 决定。信箱 producemail 中存放空消息（空位置），能取到空消息，则表示生产者进程可以生产产品，每生产一个产品，里面的空消息（空位置）数减 1。信箱 producemail 最初填满了空消息（空位置），其数量等于信箱的容量。信箱 consumemail 中存放消息（产品），有消息（产品）则表示消费者能消费，每消费一个产品，信箱 consumemail 中的消息（产品）数就减 1。信箱 consumemail 中最初是没有消息（产品）的。

生产者：从信箱 producemail 中接收空消息。若接收到空消息，则生产产品（消息），并将它发送到信箱 consumemail 中。每生产一个产品，信箱 producemail 中空消息（空位置）就减少一个。

消费者：从 consumemail 中接收消息（产品）。有消息（产品）则表示能消费，消费消息（产品）后，发送空消息到信箱 producemail 中。

生产者-消费者的通信过程可分别描述如下：

```
const int capacity=/*buffering capacity*/
    null=/*empty message*/
int i;
void producer( )
{
   message pmsg;
   while(true)
   {
      receive(producemail,pmsg);
      pmsg=produce( );
      send(consumemail,pmsg);
   }
}
void consumer( )
```

```
{
    message cmsg;
    while(true)
    {
        receive(consumemail,cmsg);
        consume(cmsg);
        send(producemail,null);
    }
}
void main( )
{
    create_mailbox(producemail);
    create_mailbox(consumemail);
    for(int i=1;i<=capacity;i++)
          send(producemail,null)
    parbegin(producer( ), consumer( ));
}
```


3．消息缓冲队列通信机制

消息缓冲队列通信机制是由 Hansen 提出的一种进程通信方式，如图 3-5 所示。其基本思想是：当发送进程要发送消息时，先根据发送的消息长度向系统申请一个消息缓冲区，接着把发送的消息写入到消息缓冲区中。然后将其挂在接收进程 PCB 所指示的消息队列的队尾。接收进程在适当的时候从其消息队列中摘下消息缓冲区，读取消息，然后再将消息缓冲作为空闲缓冲区归还给系统。

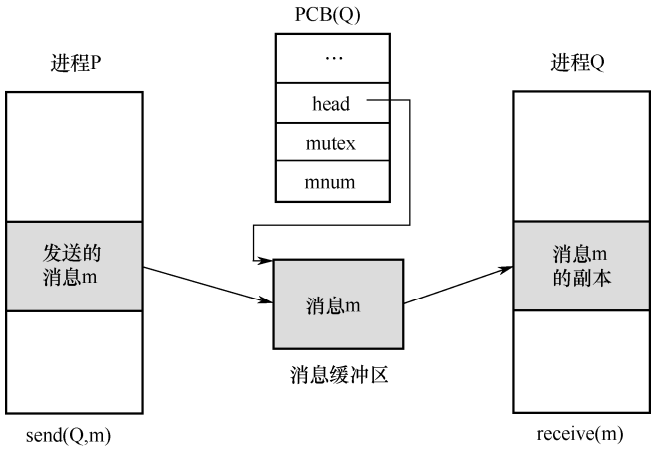

图 3-5　消息缓冲队列通信机制示例

（1）消息缓冲队列通信机制中的数据结构

① 消息缓冲区

```
struct message
          {
             int sender;        //发送者进程标识符
             int type;          //消息类型
             int size;          //消息长度
             char *text         //消息正文（含接收者标识符）
             struct message  *next //指向消息队列中下一个消息缓冲区的指针
          };
```

② PCB 结构中有关进程同步的信息

利用消息缓冲队列通信机制时，在设置消息缓冲队列的同时，还应增加用于对消息队列进行操作和实现同步的信号量，并将它们记录在进程的 PCB 中，其实现过程如下。

```
struct  PCB
          { …
             struct message  *head;   //指向该进程接收到的消息队列队首指针
             semaphore  mutex;  //消息队列是临界资源，该信号量用于它的互斥
             semaphore  mnum;  //同步信号量，其值表示消息队列中的消息数
             …
          };
```

（2）利用通信原语进行通信

① 发送原语

```
void send(Q, m)
{
   根据消息 m 的长度 size 向系统申请一个消息缓冲区;
   找到接收进程的 PCB;
   wait(mutex);
   将准备发送的消息 m 复制到新申请的消息缓冲区;
   将消息缓冲区挂在接收进程消息队列的队尾;
   signal(mutex);
   signal(mnum);
}
```

② 接收原语

```
void receive(n)
{
   wait(mnum);
   wait(mutex);
   从 head 指向的消息队列中摘下第一个消息缓冲区;
   将消息缓冲区中的信息 n 取出;
   释放消息缓冲区;
   signal(mutex);
}
```


电子工业出版社版权所有 盗版必究

小　结

现代操作系统的中心问题是多道程序设计，而实现多道程序设计的基础是并发。并发执行的多个进程间免不了形成各种各样的关系，最具代表性的关系就是同步和互斥。同步是指进程之间通过在执行时序上的某种限制而达到相互合作的约束关系，是一种直接制约关系。互斥是指进程之间彼此本无关，但是由于竞争使用同一共享资源而产生相互制约的关系，是一种间接制约关系。实现进程的同步和互斥可以使用 P 操作和 V 操作。本章通过三个典型例子生产者-消费者问题、读者-写者问题及哲学家就餐问题进行了讨论。

此外，进程之间要进行大量数据的通信，可以采取共享存储器系统、管道通信系统和消息传递系统的方式来实现通信。

习　题

3-1 什么是临界资源和临界区？

3-2 操作系统在管理可控制资源分配与使用方面，应当保证进程对临界资源的访问满足哪些要求？

3-3 互斥机制应遵循哪些基本准则？为什么？

3-4 整型信号量和记录型信号量是否完全遵循了同步机制的 4 条准则？

3-5 在生产者-消费者问题中，如果缺少了 wait 操作或 signal 操作，对执行结构会有何影响？

3-6 在生产者-消费者问题中，如果将两个 wait 操作互换，或者是将两个 signal 操作互换，结果会如何？

3-7 对于哲学家进餐问题，请利用记录型信号量设计一个不出现死锁的算法。

3-8 试用 P 操作和 V 操作描述如图 3-6 所示的程序或语句间的前趋关系。

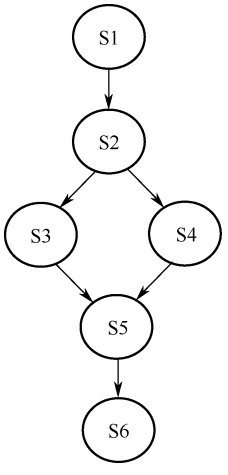

图 3-6　习题 3-8 图

电子工业出版社版权所有
盗版必究

3-9 有一个阅览室，共有 100 个座位，读者进入时必须先在一张登记表上登记，该登记表为每一位读者列一表目，包括座号和读者姓名等，读者离开时要消掉登记的信息。试用 P 操作和 V 操作描述读者进程之间的同步关系。

3-10 从读卡机上读进 *N* 张卡片，然后复制一份，要求复制出来的卡片与读进来的卡片完全一致。这一工作由三个进程 get、copy 和 put 以及两个缓冲区 buffer1 和 buffer2 完成。进程 get 的功能是把一张卡片上的信息从读卡机上读进 buffer1；进程 copy 的功能是把 buffer1 中的信息复制到 buffer2；进程 put 的功能是取出 buffer2 中的信息并从打印机上打印输出。试用 P 操作和 V 操作完成这三个进程间的并发运行的关系，并指明信号量的作用及初值。

3-11 请用信号量解决以下的“过独木桥”问题：同一方向的行人可连续过桥，当某一方向有行人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

3-12 在单处理机系统中，进程间有哪几种通信形式？

3-13 请简述消息缓冲队列通信机制的实现方法。

电子工业出版社版权所有
盗版必究