第5章 进程通信

现代操作系统均是基于进程管理的操作系统，是通过对进程的管理与控制实现多道程序并发执行的。但进程在异步并发时，由于彼此需要合作，或者由于竞争资源，使得进程在保留异步特征的同时，在特定点上还会产生速度上的相互制约。因此，若对这种速率协调关系不加以保证，进程在并发过程中可能会产生与时间有关的错误。

为了保证进程有序正确地执行，系统必须提供必要的通信机制，以保证进程间的并发。本章将讨论实现这一目的的许多不同方案。


观看视频

5.1 进程的同步与互斥

在多道程序设计环境下，任务被分解成了多个进程，而一个进程可能只完成某个任务中的一部分，多个进程的共同执行才可以使一项任务顺利完成。这就提出了以下问题：进程在执行中有哪些约束和限制，进程在执行中需要采用什么方式彼此交互信息，以保证共享信息的正确使用和进程并发的正确执行。由于这些进程共存于同一系统，它们有些是彼此无关的，但有些却并不是彼此孤立的，因此必须协调运行。这样，在进程间就产生出了某种彼此依赖或相互制约的关系。进程间的这种相互制约关系又分为同步与互斥两种情形。

5.1.1 进程合作

引起进程相互制约的第一个原因是系统中存在着一些合作进程，它们相互配合完成同一任务。因此，它们之间必然存在着逻辑上的联系，例如它们要互通消息、交换数据和某些结果等。下面通过一个简单的例子加以说明。

【例 5.1】 计算 X=fun1(y)×fun2(z)的值。其中，fun1(y)和 fun2(z)都是较为复杂的函数。

【分析】 考虑创建进程 P_1 来完成 X 的计算。为减轻 P_1 的任务，创建一个与 P_1 异步并发的进程 P_2 完成 fun2(z)的计算。显然，P_1、P_2 既是一对父子进程又是两个合作者进程，由于两者之间的直接联系，使它们在异步并发的同时在关键点上产生了某些相互制约，这种相互制约关系如图 5.1 所示。

从图 5.1 可知，进程 P_1 在完成 fun1(y)的计算之后，需在 L_1 点处检测进程 P_2 是否计算完 fun2(z)。如未计算完，则必须反复检测等待；如计算完，则取用进程 P_2 的计算结果，然后进行乘法运算。相应地，进程 P_2 在完成 fun2(z)的计算后，应该在 L_2 点处设置“计算完成”标志供进程 P_1 检测。因此，进程 P_1、P_2 之间的协同关系可描述为：P_1、P_2 异步并发，

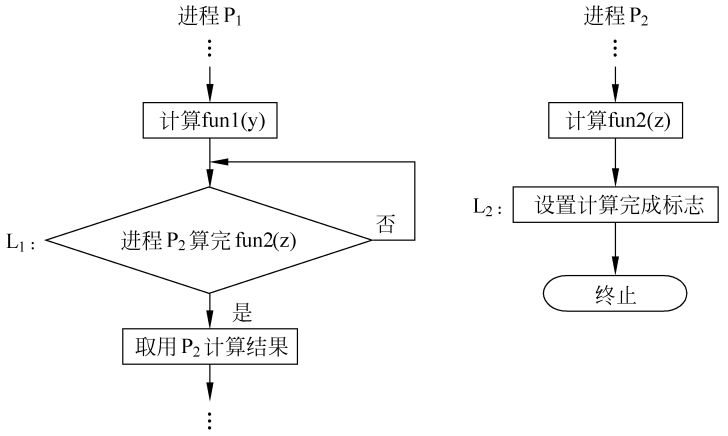

图 5.1　两个伙伴进程间的同步

但必须在某些特定的点上协调它们推进的速率，体现为当 P_1 推进至 L_1 点时，除非 P_2 已推进至 L_2 点，否则 P_1 不得不在 L_1 点处等待，直至 P_2 到达 L_2 点后才能继续向前推进。

这种合作进程之间在执行次序上的协调关系称为同步关系。进程同步(Synchronization)是指为完成共同任务的并发进程基于某个条件来协调其活动。因为需要在某些位置上排定执行的先后次序而等待和传递信息等所产生的协作制约关系。

5.1.2　共享资源

直接制约发生在彼此间有着逻辑联系的进程之间。然而，更一般的情况是，并发进程间没有逻辑上的关联，但由于它们竞争独占性资源，获得资源的进程可以继续运行，未获得资源的进程则只能暂停等待，直至其他进程释放资源后才能继续运行。这样，对同类资源的使用就使得这些进程间产生了相互制约关系，这是一种由于使用独占型资源而导致的竞争制约关系。因此，可以称为进程互斥(Mutual Exclusion)。

【例 5.2】　进程 P_1、P_2 共享一台打印机。

【分析】　显然，打印机自身的特点决定了不能让 P_1、P_2 随便使用，这样最可能发生的情况是进程 P_1、P_2 的输出结果交织在一起而难以区分。因此，需要在 P_1、P_2 的并发程序中控制对打印机的使用，使得在同一段时间内，P_1、P_2 中最多只有一个进程使用打印机。相应地，P_1、P_2 的并发程序可表示如下：

```
int x;                                  /* x为空闲打印机台数 */
x = 1;
void P1()
{   while (1)
    {   x = x - 1;                      /* 进程 P1 欲使用打印机 */
        if(x < 0) 等待打印机            /* P1 因得不到打印机而阻塞 */
        使用打印机;
        x = x + 1;                      /* 进程 P1 释放出打印机 */
        remainder section
    }
}
void P2( )
{
```

```
    while(1)
    {   x = x - 1;                          /* 进程 P2 欲使用打印机 */
        If(x < 0) 等待打印机                 /* P2 因得不到打印机而阻塞 */
        使用打印机;
        x = x + 1;
        remainder section;
    }
}
void main()
{
    cobegin
        P1();P2();                          /* cobegin 表示以下进程并发执行,至 coend 结束 */
    coend
}
```

上述程序意图控制在任何一段时间内只有一个进程使用打印机，那么该算法能否实现这一功能呢？对此进行具体分析。

假设：某时刻打印机空闲，则 x＝1。

P_1 提出使用打印机，依据上述程序，P_1 执行 x＝x－1 操作，将打印机空闲数由 1 变为 0（即表示 P_1 占据一台打印机），其后，在 if 语句中，由于 x＜0 不成立，故 P_1 不会被阻塞而顺利进入对打印机的使用，使用完毕，通过 x＝x＋1 将 x 由 0 变为 1，表示释放一台打印机；此后，若 P_2 申请打印机，同理也可获得。因此，在这种并发执行的情况下，该算法可达到正确使用打印机的目的。

但若某时刻打印机空闲，P_1 和 P_2 同时提出使用打印机，则它们都执行 x＝x－1 操作，x 的值变为－1；此后，P_1、P_2 两个进程都由于在 if 语句的判断中得出 x＜0 成立的结论，而阻塞在得不到打印机这一事件上，导致谁也无法使用空闲的打印机而陷入永远等待。显然，这一错误的发生是由进程 P_1、P_2 对共享变量 x 无控制地存取访问所导致的。即当 P_1 尚未结束对 x 的访问时，P_2 又进入了对 x 的访问。

不难看出，进程互斥关系是一种特殊的进程同步关系，即逐次使用独占型资源，也是对进程使用资源次序的一种协调。

综上所述，在多道程序环境下，对于同处于一个系统中的多个进程，由于它们共享系统中的资源，或者为完成某一个任务而相互合作，它们之间可能存在着两种形式的制约关系：进程同步（直接相互制约关系）和进程互斥（间接相互制约关系）。从广义上理解，进程同步是一个大的范畴，协作进程间的制约关系都可以称为进程同步，只是根据制约形式的不同而进一步细分。因此，可以说进程互斥是进程同步的特例，进程同步强调的是保证进程间操作的先后次序的约束，而进程互斥强调的是对共享资源的互斥访问。

5.2 临界资源与临界区

在多道程序环境中，由于存在着进程同步（直接相互制约关系）和进程互斥（间接相互制约关系），进程在运行中能否获得处理器的运行权，以及其他资源的使用权，以怎样的速度推进，都不是进程自身所能控制的。这就是并发进程的异步性。它可能带来进程对共享资源的不同访问次序，从而造成每次执行结果都不一致，这种差错往往与时间有关。为了防止这

种错误，就要协调进程的执行次序。

5.2.1　与时间有关的错误

在多道程序设计环境中，由于并发进程执行的相对速度无法相互控制，因此进程在共享相同的资源时，伴随进程间不同的相对速率，就会对同类资源产生不同的操作序列，各种与时间有关的错误就可能出现。下面通过实例来说明这种错误的表现形式。

【例 5.3】 假设一个飞机订票系统有两台终端，分别运行进程 T_1 和 T_2。若用公共变量 x 代表某次航班的订票数，则每台终端在订票后，需对 x 进行加 1 操作。相应并发程序如下：


观看视频

```
int x;                                    /* 某航班飞机订票数 */
x = 0;
void T1(x)
{   while(1)
    {   read(x);
        x = x + 1;
        write(x);
    }
}
void T2(x)
{   while(1)
    {   read(x);
        x = x + 1;
        write(x);
    }
}
void main()
{   cobegin
        T1();T2();
    coend
}
```

进程 T_1、T_2 共享软件变量资源 x，并分别对 x 做加 1 操作。这两个操作用机器语言实现时常可用下面的形式描述：

T_1：register1＝x；　　T_2：register2＝x；
　　register1＝register1＋1；　　register2＝register2＋1；
　　x＝register1；　　x＝register2；

假设：x 当前值为 5，表示某航班已订出 5 张机票。在这种情况下，如果进程 T_1 先执行左列的 3 条机器指令，然后进程 T_2 再执行右列的 3 条指令，则最后共享变量 x 的值为 7，表示 T_1、T_2 终端分别订出一张机票后，现订出机票数变为 7，这一运行结果是正确的。但是，如果按照下述顺序执行：

T_1：register1＝x；　　(register1＝5)
T_2：register2＝x；　　(register2＝5)
T_1：register1＝register1＋1；　　(register1＝6)
T_2：register2＝register2＋1；　　(register2＝6)
T_1：x＝register1；　　(x＝6)
T_2：x＝register2；　　(x＝6)

正确结果应为 x=7，但现在 x 的值是 6。这表示程序的执行不具有可再现性而产生了与时间有关的错误。这一错误的表现形式是结果不正确。为预防这种错误的发生，必须保证在任何一段时间内，T_1、T_2 两个进程只能有一个访问共享资源 x，只有当其中一个结束对 x 的访问后，另一个才能进入对 x 的访问。

从上面的例子可见，并发进程执行序列的随机性，会引起与时间有关的错误，这种错误表现为结果不唯一（如例 5.3 所示）和永远等待（如例 5.2 所示）两种情况。


观看视频

5.2.2 临界资源与临界区

由以上内容分析可知，并发进程共享的某些资源，无论是硬件资源（如打印机）还是软件资源（如飞机订票数 x），都有一个共同特点，即在任何一段时间内，只能有一个进程访问这种资源。换言之，诸进程在共享这类资源时必须互斥地对其进行访问。

1. 临界资源

我们把一次仅允许一个进程使用的资源称为临界资源（Critical Resource，CR）。许多物理设备都属于临界资源，如打印机、绘图仪等。除物理设备外，还有许多变量、数据、文件等都可由若干进程所共享，它们也属于临界资源。

2. 临界区

我们把每个进程中访问临界资源的那段程序代码称为临界区（Critical Section，CS）。显然，进程异步并发，但为了避免发生与时间有关的错误，诸进程必须互斥地进入自己的临界区，互斥地访问临界资源。即当一个进程进入临界区使用临界资源时，其他进程必须等待；当占有临界资源的进程退出临界区后，另一个进程才被允许进入其临界区。此即进程之间的互斥关系。

为避免两个进程同时进入临界区，可以采用软件解决方法，也可在系统中设置专门的同步机构来协调进程。

为此，针对每个进程的程序，从概念上将其分为两部分：一部分是涉及临界资源的临界区；另一部分是其余的不涉及临界资源的非临界区部分。

显然，为保证临界资源的互斥使用，每个进程在进入其临界区前，应对欲访问的临界资源进行检查，只有当临界资源未被访问时，该进程才可进入临界区，并设置临界资源正被访问的标志；如果此刻临界资源正被其他进程访问，则该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，把这段代码称为进入区（Entry Section）代码。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，用于表明进程已释放相应临界资源。这样，一个访问临界资源的循环进程可描述如下：

```
while(1)
  {   entry section
      critical section;          //临界区操作
      exit section
      remainder section;         //继续其余部分
}
```

5.2.3 同步机构设计准则

对于保证进程互斥与同步的同步机构而言，无论其采用何种技术实现，均应遵循以下 4

条设计准则。

1. 空闲让进

当有进程要求进入临界区，而此刻无其他进程处于临界区内，即相应的临界资源处于空闲状态，则应该允许请求者立即进入临界区，以有效地利用临界资源。

2. 忙则等待

当已有进程进入了临界区，意味着临界资源正被访问，则所有其他试图进入临界区的进程必须等待，这是为了保证诸进程互斥地访问临界资源。

3. 有限等待

对要求访问临界资源的进程，应保证该进程能在有限的时间内进入自己的临界区。换言之，进程之间不应因为争夺临界资源而相互阻塞，陷入“死等”状态。

4. 让权等待

当一个进程不能进入临界区时，应立即释放处理器，以避免进程陷入“忙等”状态。这一准则的设立是为了提高处理器的有效利用率。

这 4 条准则非常实用好记，举个例子，大家都用过 ATM 取款机，每个 ATM 取款机的小房间都是临界资源。只有当此处无人使用时你才可以进去，这叫“空闲让进”；如果门锁着，有人使用，那么你会排队等待在外，这就是“忙则等待”；当然，你会根据情况有限地等待，如果里面的人长时间不出来，你就要灵活机动地换台 ATM 取款机，不会一直死等，这是“有限等待”；而如果当你终于排队等到了 ATM 取款机时，却发现忘记带银行卡了，你只好马上出来，将这台空闲资源让权给其他人使用，这就是“让权等待”。通过这个例子，可以看到进程的同步与互斥问题在生活中很多地方都存在，后面讲授的“信号量机制”在日常生活的交通管理中也会遇到。

5.3 临界区管理的软件方法

对临界区互斥访问技术的研究始于 20 世纪 60 年代，早期的解决方法用软件方法实现，也出现了一些比较具有代表性的算法。下面将列举一些。

【例 5.4】 有两个并发进程 P_0 和 P_1，互斥地共享单个临界资源(如磁带机或某共享数据)。P_0 和 P_1 是循环进程(指进程执行一个无限循环程序)，每次使用资源的时间是一个有限的时间段。

【分析】 由题意，并发程序的结构如下：

```
void main()
{       common variable declaration;
        cobegin
            P0();P1();
        coend
}
```

1. Peterson 算法

Peterson 算法是一个实现互斥锁的并发程序设计算法，可以控制两个线程访问一个共享的单用户资源而不发生访问冲突。1981 年，Gary L. Peterson 提出该算法来解决进程互斥进入临界区的问题。该算法使用两个控制变量 flag 与 turn。其中 flag[n]的值为真，表示

ID号为n的进程希望进入该临界区；变量turn用于保存有权访问共享资源的进程的ID号。

```
Boolean flag[2];
int turn;
void P0()
{
    while(1)
    {
        flag[0] = true;
        turn = 1;
        while(flag[1]&&turn == 1);     /* 条件满足时,循环体是空语句,无须做什么,等待 */
        /* 若 flag[1]为 false,则进程 P0 进入临界区;若 flag[1]为 true,则 P0 循环等待,只有 P1
退出临界区,P0 才可进入 */
        critical section               /* 访问临界区 */
        flag[0] = false;               /* 访问临界区完成,进程 P0 释放出临界区 */
        remainder section              /* 其余程序段 */
    }
}
void P1()
{
    while(true)
    {
        flag[1] = true;
        turn = 0;
        while(flag[0]&&turn == 0);     /* 条件满足时,循环体是空语句,无须做什么,等待 */
        critical section               /* 访问临界区 */
        flag[1] = false;               /* 访问临界区完成,P1 释放出临界区 */
        remainder section              /* 其余程序段 */
    }
}
void main()
{
        flag[0] = flag[1] = false;
        cobegin(P0,P1);                /* 并发运行 P0,P1 */
}
```

若要证明Peterson算法是正确的，则须证明该算法满足解决临界区问题的3个准则：空闲让进、忙则等待、有限等待。需要说明的是，第4个准则“让权等待”属于较高要求，在早期的解决方案中均未对此提出要求，因为这样的做法虽会影响系统效率，但不影响临界区问题的解决。

证明如下：假设初始时临界区没有进程在执行，现在要么1个进程要求进入，要么2个进程要求进入。针对第一种情况，例如进程P_0要求进入，那么flag[0]=true，turn=1，此时flag[1]=false，因此进程P_0中while语句的判断条件为false，进程P_0顺利进入临界区，符合“空闲让进”准则。针对第2种情况，由于两个进程都希望进入临界区，都可能更改flag[0]=flag[1]=true，但是turn的值只可能取0或者1，不可能同时取两个值，因此这两个进程不可能同时进入临界区，只能有一个进程率先进入。假设是进程P_0进入，现在临界区中操作，另一个进程P_1希望进入临界区时，由于flag[0]=true，turn=0，就会使得进程P_1中while语句的判断条件始终为true，进程P_1处于“忙等”状态而无法进入临界区，因此符合“忙则等待”准则。直到进程P_0结束临界区的操作，修改flag[0]=false，退出临界区，此时

P_1 的 while 语句的判断条件为假，从而 P_1 可以进入临界区，也符合“有限等待”准则。因此，Peterson 算法保证了临界区中始终只能有一个进程执行，从而保证进程对临界资源进行互斥访问。

2. Dekker 算法

Dekker 算法是由荷兰数学家 T. Dekker 提出的，此方法是为每个进程设置一个标志位 flag[i]，当该位为 1 时，表示进程 P_i 要求进入临界区或已在临界区中执行。另外，还设置了一个共享变量 turn 用以指出应该由哪一个进程进入临界区，若 turn=i，则应该由 P_i 进入临界区。其程序如下：

```
int flag[2];
int turn;
flag[0] = flag[1] = 0;
turn = k;                                   /* k 为 0 或 1 */
while(1)
{   flag[i] = 1;
    while(flag[j])
    {   if(turn == j)
        {   flag[i] = 0;
            while(turn == j) skip;
            flag[i] = 1;
        }
    }
    critical section;                       //临界区操作
    turn = j;
    flag[i] = 0;
    remainder section;                      //继续其余部分
}
```

类似于 Peterson 算法，读者可以自行证明 Dekker 算法的正确性。

3. Dijkstra 算法

可把 Dekker 算法扩充到解决 n 个进程的临界区问题，这一算法是由 Dijkstra 于 1965 年提出的，故称为 Dijkstra 算法。该算法使用的公共数据结构是：

```
int flag[n];
int turn;
flag[0] = flag[1] = … = flag[n - 1] = 0;
turn = k;
```

flag 数组所有元素的初值均为 0，1 表示相应进程处于申请进入临界区的状态，2 表示相应进程正在临界区中执行。turn 的初值 k 可为 0～$n-1$ 中的任意值。

进程 P_i（$i=0,1,\cdots,n-1$）的结构如下：

```
int  j;
while(1)
{   do
{   flag[i] = 1;
        while(turn!= i) if(flag[turn] == 0) turn = i;
        flag[i] = 2;
        j = 0;
        while(j < n && (j == i or flag[j]!= 2) j++;
    }while(j < n)
```

```
    critical section;                    //临界区操作
    flag[i] = 0;
    remainder section;                   //继续其余部分
}
```

可以证明这个算法也是正确的。

首先它保证了进程互斥地进入临界区，因为仅当 flag[j]!=2 对所有 j≠i 都成立时，进程 P_i 进入临界区。因此，只有 P_i 的 flag[i]=2，也就是说只有 P_i 在临界区内。

这个算法也不会造成进程间相互阻塞，这是因为：

(1) 任何进程，只要正在执行临界区代码，它的状态就不会是 0，即 flag[i]≠0。

(2) 若某个进程 P_i 的状态是 flag[i]=2，这并不意味着 turn=i，因为几个进程可能同时进入 do 循环，并且都发现 flag[turn]=0，于是这些进程相继把 turn 指针改为指向自己。

(3) 第一个进程把 turn 指针改为指向自己以后，那些尚未查询过 flag[turn]=0 是否成立的进程再也不可能把 turn 指针指向自己，即不可能执行 turn=i 语句。

(4) 若$\{P_1,P_2,\cdots,P_m\}$是同时进入 do 循环的一批进程的集合，由于它们都发现 flag[turn]=0，从而把指针都指向了自己，并将自己的状态改为 flag[i]=2。但 turn 指针最终是指向最后一个改变 turn 指针值的进程 P_k 的($1\leqslant k\leqslant m$)。对于所有属于这一集合的进程 $P_i\in\{P_1,P_2,\cdots,P_m\}$，由于在执行第 3 个 while 语句时，发现除自己外还有其他进程的状态也是 flag[j]=2，而被迫返回执行 do 循环语句，于是只有 P_k 顺利进入临界区，所以进程间不会互相阻塞，并且是在有限的时间内就进入了临界区。

从以上算法不难看出，软件解决方法不仅复杂，而且难以理解。下面介绍临界区问题的硬件解决方法。

5.4 临界区管理的硬件方法

完全利用软件方法来解决诸进程互斥进入临界区的问题，有一定难度且有很大局限性。实际上，现代计算机提供了一些特殊的硬件指令，这些指令允许对一个字中的内容进行检测与修正，或交换两个字的内容等。这些特殊指令也可以用来解决临界区问题。

实现进程互斥的硬件方法的优点是可以保证共享变量的完整性和正确性，并且简单、有效。在对临界区进行管理时，可以想象成进入临界区要先看看门上的锁，如果锁是打开的，那么允许进入，如果锁是关闭的，那么进程要在外面等待。每个想要进入临界区的进程，都必须先对锁进行测试，当锁未开时，在外等待，当锁打开时，则进入临界区，并立即把锁关闭，以阻止其他进程进入临界区。显然，为了防止多个进程同时测试到“锁开”，测试和关锁的操作必须是连续的，不允许分开进行。

5.4.1 关中断

关中断是实现互斥的简单方法之一。在进程进入临界区时关中断，在进程退出临界区时开中断。中断被关掉后，时钟中断也被屏蔽。进程在临界区执行期间，计算机系统不响应中断，因此不会引发调度，不会发生进程或线程切换。这样进程的执行不会被打断，因此采用开中断、关中断的方法就可以保证并发进程互斥地进入临界区。

关中断的方法简单有效，但是关中断的方法存在着诸多缺点：关中断时间过长，会影响

系统效率,限制了处理器并发执行程序的能力。关中断方法也不适用于多 CPU 系统,因为在一个处理器上关中断,并不能防止进程在其他处理器上执行相同的临界段代码。因此,关中断方法并不适合作为通用的互斥机制。

5.4.2 TS 指令

在测试与设置方法中,针对一个临界资源,设置一个变量 lock 来代表该临界资源的状态,用 lock 为"假"表示资源可用,用 lock 为"真"表示资源已被占用。变量 lock 也被形象地称为"锁"。据此,进程使用临界资源的操作应按如下 3 步进行。

(1) 测试 lock 的值,若 lock 为"真",表示资源已被占用,则不断测试等待;若 lock 为"假",表示资源可用,则置 lock 为"真",以排斥其他进程进入临界区。这一过程可形象地称为"关锁"。

(2) 进入临界区,访问临界资源。

(3) 退出临界区,置 lock 为"假",以便其他进程能使用临界资源。这一过程称为"开锁"。

在早期同步机构中,测试与设置技术通过操作系统内核提供的开/关锁原语来实现。为提高系统运行效率,依据软件固化的思想,现代操作系统中广泛采用硬件测试与设置指令(Test-and-Set,即 TS 指令)来实现这一功能。

TS 指令的功能描述为:

```
int TS(int lock)
{   TS = lock;
    lock = 1;
    return(TS);
}
```

必须注意的是,以上是 TS 指令的功能描述,并非软件实现定义。事实上,TS 指令的实现是由硬件系统直接完成的,并由硬件保证其原子操作性能。

如果机器支持 TS 指令,则互斥问题可以通过设置一个整型变量 lock(初值为 0),用下列方法解决:

```
while(1)
{   while(TS(lock)) skip;
    critical section                    //临界区操作
    lock = 0;
    remainder section;                  //继续其余部分
}
```

上述描述是利用 TS 指令实现互斥的通用模型,借助这一模型可以方便地编写出实现进程互斥的程序。

【例 5.5】 进程 P_1、P_2 共享一台打印机,试用硬件指令机制编写它们正确互斥的程序。

【分析】 对于这一互斥问题,可使用上述互斥模板编写出 P_1、P_2 的并发程序:

```
int lock;
lock = 0;
void P1()
{       while(1)
        {   while(TS(lock)) skip;      /* 进程 P1 试图进入临界区 */
            使用打印机;                /* 进程 P1 进入临界区 */
```

```
            lock = 0;
            remainder section;
        }
}
void P2()
{       while(1)
        {   while(TS(lock)) skip;      /* 进程 P2 试图进入临界区 */
            使用打印机;                /* 进程 P2 进入临界区 */
            lock = 0;
            remainder section;
        }
}
void main()
{   cobegin
        P1();P2();
    coend
}
```

5.4.3 对换指令

对换指令在 Intel 80x86 中又被称为 XCHG 指令，用于交换两个字的内容。其处理过程描述为：

```
void swap(boolean a,boolean b)
{
    boolean temp;
    temp = a;
    a = b;
    b = temp;
}
```

用对换指令可以简单有效地实现进程互斥。方法是为每个临界区设置一个全局布尔变量 lock，其初值为 false，在每个进程中再设置一个局部布尔变量 key，使用 swap 指令与 lock 进行数值交换，以此来判断 lock 的取值。只有当 key 为 false 时，进程才可以进入临界区操作。进程执行 swap 指令的循环过程描述如下：

```
boolean lock = false;
void P1()
{
    boolean key = true;
    do{
        swap(key,lock);
        }while (key);
    critical section                    //临界区操作
    swap(key,lock);
}
```

用硬件指令方法实现进程互斥有着明显的优点。

(1) 方法简单，行之有效。

(2) 可以用于多重临界区的情况。每个临界区可以定义自己的共享变量。

但是也有明显的缺点，那就是硬件方法一般采取“忙等待”的简单方法。如果在单处理器系统中，一个进程试图进入处于“忙”状态的临界区，则该进程只能不断测试临界区的状

态，这将浪费大量的处理器时间。此外，硬件指令机制难以解决较为复杂的进程同步问题。

5.5 信号量机制与管程

1965年，荷兰著名的计算机科学家Dijkstra提出了一种称为信号量的同步机制。其基本原理是在多个进程间使用简单的信号来实现同步。一个进程被强制停止在一个特定的地方，直到收到一个专门的信号。这个信号就是信号量(Semaphore)，其工作方式类似于铁路交通管理中的信号灯。

在操作系统中，信号量是联系某类临界资源的数据结构，信号量的不同取值表示临界资源的不同状态。进程通过对信号量实施合法操作实现交换彼此的控制信息、协调彼此的执行顺序或互斥使用临界资源的目的。

作为一种卓有成效的进程同步工具，在长期且广泛的应用中，信号量机制得到了很大的发展，它从整型信号量经记录型信号量，发展为信号量集机制，现在已被广泛地应用于单处理器、多处理器系统以及网络操作系统中。

Dijkstra将操作信号量的两个原语命名为P操作和V操作(因为在荷兰语中，“通过”叫passeren，相当于英语中的pass，而“增加”叫verhoog，相当于英语中的increment。P、V即来源于这两个单词的第一个字母，P、V操作因此得名)，本书为与现代操作系统教材的称谓保持一致，把P操作称作wait操作，V操作称作signal操作。

按照信号量的用途的不同，可把信号量分为两类：互斥信号量和同步信号量。

1. 互斥信号量

其初值仅允许取值为0或1，主要用于控制进程互斥地进入临界区，也称公用信号量。

2. 同步信号量

其初值为初始资源数，主要用于控制进程间的同步运行，也称私有信号量。

5.5.1 整型信号量

整型信号量是Dijkstra最初定义的信号量形式，它将信号量s定义为一个整型变量，除初始化外，该信号量s的值仅能通过两个标准的原子操作(Automatic Operation)wait(s)和signal(s)加以访问。

wait(s)和signal(s)操作可描述为：

```
void wait(s)
{   while(s<=0);                    /* do no operation,什么都不做 */;
    s--;
}
void signal(s)
{   s++;
}
```

整型信号量的不足之处在于，当一个进程执行wait(s)操作时，若发生s≤0的情况，则该进程必将不断循环测试，陷入“忙等待”。因此，用整型信号量作为同步机制的实现技术不符合让权等待的设计准则。

5.5.2 结构型信号量

结构型信号量是一种不存在“忙等待”问题的进程同步机制。当一个进程对信号量进行wait操作而不能通过时，该机制便使其让权等待。因而在采用这一策略后，会出现多个进程同时等待访问同一临界资源的情况。为此，在结构型信号量中，除需要一个表示资源数量的整型变量value外，还需为每个信号量增加一个进程链表指针L，用以链接所有等待该信号量的进程的PCB。


观看视频

一个结构型信号量可定义如下：

```
struct semaphore
{    int value;
     list of process *L;
};
```

wait(s)和signal(s)原语可描述为：

```
void wait(struct semaphore s)
{    s.value--;
     if(s.value<0) block(s.L);
}
void signal(struct semaphore s)
{    s.value++;
     if(s.value<=0) wakeup(s.L);
}
```

结合信号量s的含义，分析wait(s)和signal(s)操作，可得如下结论。

(1) 从物理概念上，作为联系某类临界资源的物理实体，信号量的值s.value与相应临界资源的数量有关。其中，s.value的初值表示系统中该类资源的可用数量。当s.value≥0时，表示某时刻资源的空闲数量；当s.value<0时，|s.value|表示因为得不到资源而被阻塞的进程数量。

(2) 执行一次wait操作，意味着进程请求一个单位的资源，因此描述为s.value--。当s.value<0时，进程由于得不到资源自行阻塞，此时将引起进程重新调度。

(3) 执行一次signal操作，意味着进程释放一个单位的资源，因此描述为s.value++。若s.value≤0，表示链表s.L中仍有因请求该资源而阻塞的进程，此时把s.L中的第一个进程唤醒并将其转移到就绪队列中。

(4) 必须指出，在信号量机制的实现中，一个重要的问题是要保证对信号量操作的原子性。若这种原子性得不到保证，则会引起与时间有关的错误。

5.5.3 AND型信号量

在结构型信号量机制中，wait(s)操作只能申请一个单位的某种资源，signal操作也只能释放一个单位的某种资源。而在有些应用场合中，一个进程需要同时获得多种临界资源后才能执行其任务。若要通过一个原子操作来获得或释放多种临界资源，则需要使用AND同步机制。它联系着两个原语：Swait和Ssignal。其定义如下：

```
Swait(S1,S2,…,Sn)
{    if(S1>=1&& S2>=1…&& Sn>=1 )
```

```
        for(i = 1;i < = n;i++) Si: = Si - 1;
    else
        Place the process in the waiting queue associated with the first Si found
        with Si < 1, and set the program count of this process to the beginning of swait
operation;
}
Ssignal(S1,S2,…,Sn)
{   for(i = 1;i < = n;i++)
    {   Si: = Si + 1;
        Remove all the process waiting in the queue associated with Si into
        the ready queue.
    }
}
```

5.5.4 信号量集

在前面所述的结构型信号量机制中，wait(S)或 signal(S)操作仅能对信号量施以加 1 或减 1 操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。当一次需要 N 个资源时，要进行 N 次 wait(S)操作，这显然是低效的。此外，在某些情况下，为确保系统的安全性，当所申请的某类资源剩余数量低于某一下限值时，还必须进行管制，不予分配。因此，当进程申请某类临界资源时，在每次分配之前，都必须测试查看资源的数量，确定是否允许分配。

基于以上原因，可以对 AND 型信号量加以扩充。对进程所申请的所有资源以及每类资源不同的需求量，在一次原子操作中就完成申请或释放。进程对信号量 S_i 的测试值不再是 1，而是该资源的分配下限值 t_i，即要求 $S_i \geqslant t_i$，否则不予分配。一旦允许分配，进程对该资源的需求值为 d_i，表示资源占有量。进行 $S_i = S_i - d_i$ 的操作，而不是简单的 $S_i = S_i - 1$。由此形成“信号量集”机制。对应的 Swait 和 Ssignal 格式如下：

```
Swait(S1,t1,d1,…,Sn,tn,dn);
Ssignal(S1,t1,d1,…,Sn,tn,dn);
```

对于“信号量集”还有下面几种特殊情况。

(1) Swait(S,d,d)。此时在信号量集中只有一个信号量 S，但允许它每次申请 d 个资源，当现有资源数少于 d 时，不予分配。

(2) Swait(S,1,1)。此时的信号量集已退化成一般的结构型信号量(当 S>1 时)或者互斥信号量(当 S=1 时)。

(3) Swait(S,1,0)。这是一种很特殊且很有用的信号量操作。当 S≥1 时，允许多个进程进入某特定区域；当 S 变为 0 后，将阻止任何进程进入特定区域。换言之，它相当于一个可控开关。

5.5.5 管程机制

信号量机制虽然可以解决进程的同步与互斥问题，但是由于每个要访问临界资源的进程都必须自备同步操作 wait(S)和 signal(S)，因此使得大量的同步操作散布在各个进程中。这不仅使系统不易管理，更容易由于同步操作使用不当造成死锁。因此，在 1974 年和 1977 年，Hore 和 Hansen 提出了新的进程同步工具——管程(Monitors)。

引入管程机制的目的如下。

（1）把分散在各进程中的临界区集中起来进行管理。

（2）防止进程有意或无意地违法同步操作。

（3）便于用高级语言来书写程序，也便于验证程序的正确性。

那么什么是管程呢？由于系统中的各种硬件资源和软件资源均可用数据结构抽象地描述其资源特性，即用少量信息和对该资源所执行的操作来表征该资源，而忽略它们的内部结构和实现细节。因此，可以利用公共数据结构来抽象地表示系统中的共享资源，并将对该公共数据结构实施的特定操作定义为一组过程。管程就是由局部于自己的若干公共变量及其说明和所有访问这些公共变量的过程所组成的软件模块。进程对共享资源的申请、释放以及其他操作都必须通过管程来实现。对于多个请求访问同一资源的并发进程，根据资源的情况接受或阻塞，确保每次只有一个进程进入管程，从而有效地实现进程互斥。

管程的语法描述如下：

```
Monitor monitor_name                    /* 管程名 */
{
    Share variable declarations;        /* 共享变量说明 */
    Condition declarations;             /* 条件变量说明 */
    Public:                             /* 能被进程调用的过程 */
        void P1(…)                      /* 对数据结构操作的过程 */
        {…}
        void P2(…)
        {…}
        …
        void Pn(…)
        {…}
        …
        {                               /* 管程主体 */
            Initialization code;        /* 初始化代码 */
            …
        }
}
```

可以看出，管程中包含面向对象的思想，它将表达共享资源的数据结构及其对数据结构操作的过程，都封装在一个对象内部，并封装了同步操作，对进程隐蔽了同步细节，简化了同步功能的调用界面。用户编写并发程序如同编写顺序执行的程序一样。

从以上管程的语法定义中可以看出，管程是由 4 部分组成的。

（1）管程的名称。

（2）局部于管程的共享数据结构说明。

（3）对该数据结构进行操作的一组过程。

（4）对局部于管程的共享数据进行初始化的语句。

管程是一种程序设计语言的结构成分，它和信号量具有同等的表达能力。从语言角度来看，管程具有以下特性。

（1）共享性：管程可被系统范围内的进程互斥访问，属于共享资源。

（2）安全性：管程的局部变量只能由管程的过程访问，不允许进程或其他管程直接访问，管程也不能访问其局部变量以外的变量。

(3) 互斥性：多个进程对管程的访问是互斥的。任一时刻，管程中只能有一个活跃的进程。

(4) 封装性：管程内的数据结构是私有的，只能在管程内使用，管程内的过程也只能使用管程内的数据结构。进程通过调用管程的过程使用临界资源。

管程在 Java 中已实现，有兴趣的读者可以自行扩展阅读。

5.6　用信号量机制实现进程的互斥与同步

利用信号量机制可以实现进程的互斥与同步。下面分别论述。

5.6.1　用信号量机制实现进程的互斥


观看视频

信号量机制能用于解决进程之间的互斥问题。n 个进程共享一个用于互斥的信号量 mutex，将其初值置为 1，表明任何时刻只能有一个进程获得临界资源。然后将各进程的临界区 CS 置于 wait(s)和 signal(s)操作之间即可。

利用信号量实现互斥的一般形式可描述如下：

```
while(1)
{    wait(mutex);
     critical section;
     signal(mutex);
}
```

【例 5.6】 进程 P_1、P_2 共享一台打印机，试使用信号量机制编写 P_1、P_2 并发执行的程序。

【分析】 设置一个互斥信号量 mutex 以控制打印机的互斥使用，程序如下：

```
struct semaphore mutex;                  /* 定义一个信号量 mutex */
mutex.value = 1;                         /* 表示初始时系统有一台打印机 */
void P1()
{    while(1)
     {    wait(mutex);                   /* P1 请求使用打印机 */
          使用打印机;                    /* P1 进入临界区 */
          signal(mutex);                 /* P1 释放打印机 */
          remainder section;
     }
void P2()
{    while(1)
     {    wait(mutex);                   /* P2 请求使用打印机 */
          使用打印机;                    /* P2 进入临界区 */
          signal(mutex);                 /* P2 释放打印机 */
          remainder section;
     }
void main()
{    cobegin
          P1(); P2();
     coend
}
```

5.6.2　用信号量机制实现进程的同步


观看视频

信号量机制也能用于解决进程间的各种同步问题。但只有分析出合作进程间具体的同

步关系，才能编写出正确的同步程序。

【例 5.7】 苹果、橘子问题：桌上有一个空盘，可以放一个水果。爸爸削好苹果后，可向盘中放入苹果，妈妈剥好橘子后，可向盘中放入橘子。儿子专等吃盘中的橘子，女儿专等吃盘中的苹果。规定一次只能放一种水果，试用信号量机制写出爸爸、妈妈、儿子、女儿正确同步的程序。

【分析】 在整个过程中，爸爸、妈妈、儿子、女儿间的同步关系如下。

(1) 只有盘中空时，爸爸、妈妈才可分别向盘中放入苹果或橘子。

(2) 只有盘中有苹果时，女儿才可取出苹果。

(3) 只有盘中有橘子时，儿子才可取出橘子。

因此，在本例中，应该设置 3 个信号量，plate 表示盘子是否为空，其初值为 1；apple 表示盘中是否有苹果，其初值为 0；orange 表示盘中是否有橘子，其初值为 0。则爸爸、妈妈、儿子、女儿的并发程序如下：

```
/* 请注意,为了编写方便,在信号量定义的同时赋予初值,简写为以下形式.后续例题中类似 */
struct semaphore plate = 1,apple = 0,orange = 0;
void father()
{   while(1)
    {   prepare an apple;
        wait(plate);                    /* 申请向盘中放入苹果 */
        put an apple in the plate;
        signal(apple);                  /* 向女儿发出可以取苹果的信号 */
    }
}
void mother()
{   while(1)
    {   prepare an orange;
        wait(plate);                    /* 申请向盘中放入橘子 */
        put an orange in the plate;
        signal(orange);                 /* 向儿子发出可以取橘子的信号 */
    }
}
void son()
{   while(1)
    {   wait(orange);                   /* 申请取出橘子 */
        get an orange from the plate;
        signal(plate);                  /* 向爸爸、妈妈发出盘子空的信号 */
        eat…;
    }
}
void daughter()
{   while(1)
    {   wait(apple);                    /* 申请取出苹果 */
        get an apple from the plate;
        signal(plate);                  /* 向爸爸、妈妈发出盘子空的信号 */
        eat…;
    }
}
void main()
{   cobegin
        father();mother();son();daughter();
```

```
    coend
}
```

【例 5.8】 公共汽车上的司机和售票员各司其职，为完成运送乘客的任务而相互配合。他们的活动如图 5.2 所示。试用信号量机制编写司机和售票员同步的程序。

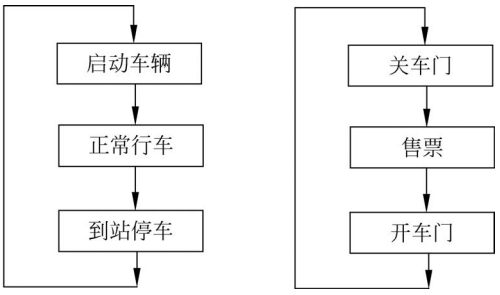

图 5.2　司机与售票员同步关系

【分析】 在汽车行驶过程中，司机和售票员之间的同步关系如下：

(1) 只有等售票员关好车门后，向司机发出开车信号，司机才可以启动车辆。

(2) 只有等司机到站停车后，售票员才可以打开车门。


观看视频

因此，在本例中，应设置两个信号量：s1 和 s2。s1 表示是否允许司机启动汽车，其初值为 0；s2 表示是否允许售票员打开车门，其初值也为 0。则司机和售票员的并发程序为：

```
struct semaphore s1 = 0,s2 = 0;
void driver()
{   while(1)
    {   wait(s1);                    /* 司机申请启动汽车 */
        启动车辆;
        正常行车;
        到站停车;
        signal(s2);                  /* 向售票员发出可以打开车门的信号 */
    }
}
void busman()
{   while(1)
    {   关车门;
        signal(s1);                  /* 向司机发出可以开车的信号 */
        售票;
        wait(s2);                    /* 售票员申请打开车门 */
        开车门;
        上下乘客;
    }
}
void main()
{   cobegin
        driver();busman();
    coend
}
```

5.7　经典进程同步问题

信号量机制可以方便地建立起进程互斥的通用模型。但使用信号量实现进程同步时，

由于合作进程内在同步关系的不同以及它们共享资源的差异，很难找到一个统一的通用模型来描述所有的同步问题，这也使得进程同步成为多道程序环境下一个重要且相当有趣的问题，吸引了众多学者进行研究，由此产生了一系列经典的同步问题。本节将对几个较为著名的同步问题加以讨论。

观看视频

5.7.1 生产者-消费者问题

生产者-消费者问题是对合作进程内在关系的一种抽象。例如，输入进程与计算进程在合作时，输入进程可比作生产者，计算进程可比作消费者；而对计算进程与打印进程而言，计算进程显然是生产者进程，而负责打印计算结果的打印进程则是消费者进程。因此，生产者-消费者问题是同步问题中最具有代表性和实用价值的一类问题。

生产者-消费者问题可描述为：一组生产者(Producer)和一组消费者(Consumer)通过一个容量为 n 的有界缓冲区(Buffer)共同完成生产和消费任务。每个缓冲区可存放一个产品，生产者负责向其中投放产品(Product)，而消费者负责消费其中的产品，其合作关系如图 5.3 所示。灰色部分表示已投放产品的缓冲区，其余为未投放产品的缓冲区。in 为生产者投放产品的位置，out 为消费者取出产品的位置。

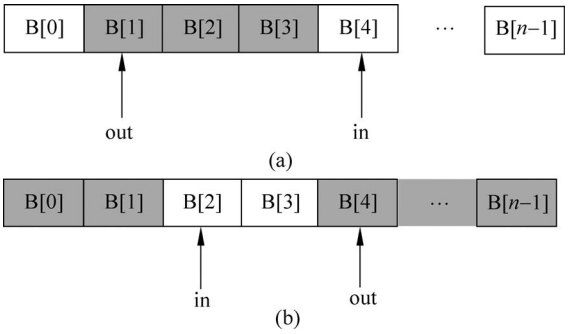

图 5.3 生产者-消费者问题

由于生产者与消费者共享的缓冲区容量为 n，最多只可存放 n 个产品。因此，生产者进程与消费者进程在运行过程中必须遵循两条限制。

(1) 生产者进程生产产品的速度不能超过缓冲区的有限容量，否则会造成产品溢出。

(2) 消费者进程消费产品的速度不能快于生产者进程的生产速度，否则会造成向空的缓冲区中提取产品。

由此得出，生产者与消费者进程应满足以下同步规则：

(1) 若生产者企图把产品放入已满的缓冲区，则必须阻塞等待，直到消费者从缓冲区中取走一个产品后将其唤醒。

(2) 若消费者企图从空缓冲区取走产品，则必须阻塞等待，直到生产者放入一个产品后将其唤醒。

为实现上述同步关系，可设立两个信号量：empty，表示某时刻空缓冲区的数目，其初值为缓冲区数量 n；full，表示某时刻放有产品的缓冲区数，初值为 0。

此外，对生产者和消费者进程来说，缓冲区是临界资源，任何一个进程对某个缓冲区的存或取必须和其他进程互斥执行。可利用互斥信号量 mutex 实现对缓冲区的互斥访问。

为保证产品依次存入和取走，还应为缓冲区设置一对读写指针 in 和 out。

生产者-消费者问题的同步模型描述如下：

```
Struct semaphore mutex1 = 1, mutex2 = 1, empty = n, full = 0;
product buffer[n];                    /* 假设缓冲区是 product 型数组 */
int in = 0, out = 0;                  /* in 为放置产品位置指针,out 为取出产品位置指针 */
void producer ()
{
    message m;
    while (1)
    {
        produce a new product in m;
        wait(empty);                  /* 申请空缓冲区,若无则等待 */
        wait(mutex1);                 /* 有空缓冲区,诸生产者进程互斥使用缓冲区 */
        buffer[in] = m;               /* 把一个产品送入缓冲区 */
        in = (in + 1) % n;            /* 把放入产品的指针移向下一个位置 */
        singal(mutex1);               /* 唤醒正在等待放入产品的其他生产者 */
        singal(full);                 /* 唤醒消费者 */
    }
}
void Consumer (void)
{   message m;
    while(1)
    {
        wait(full);                   /* 申请一个产品,若无则等待 */
        wait(mutex2);                 /* 有产品,诸消费者进程互斥使用缓冲区 */
        m = buffer[out];              /* 从缓冲区取出一个产品 */
        out = (out + 1) % n;          /* 把消费产品的指针移向下一个位置 */
        signal(mutex2);               /* 唤醒正在等待消费的其他消费者 */
        signal(empty);                /* 唤醒生产者 */
        consume product in m;
    }
}
void main()
{   cobegin
        Producer(); consumer();
    coend
}
```

应当注意，无论在生产者进程中还是在消费者进程中，signal 操作的次序都无关紧要。在以上程序中使用了两个信号量 mutex1 和 mutex2，说明有两个信号量分别控制生产者和消费者指针的移动，也就是可以同时执行放入产品和取走产品的操作。而如果只定义 1 个信号量 mutex，控制同一时刻只能执行放入产品或者取走产品的一个操作，那么 wait 操作的次序不能颠倒，否则将可能造成死锁。这一点留待读者自己分析。

5.7.2 哲学家就餐问题


观看视频

1965 年，Dijkstra 提出并解决了一个哲学家就餐的同步问题。从那时起，每个发明新同步原语的人都希望通过解决哲学家就餐问题来展示其同步原语的精妙之处。这个问题可简单描述如下：5 个哲学家一起思考并用餐。他们共享一张放有 5 把椅子的圆桌，每人占用 1 把椅子。圆桌上有 5 盘通心粉，相邻两只盘子间有一把叉子。哲学家思考(think)问题时并

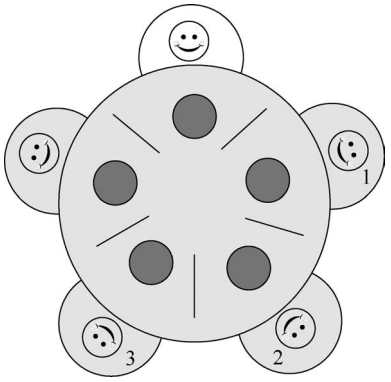

图 5.4　哲学家就餐问题

不影响他人。当他感到饥饿时便去吃通心粉，吃通心粉时需要同时使用两把叉子，规定哲学家只能取用其左边和右边的叉子，进餐完毕后，放下叉子继续思考，如图 5.4 所示。

哲学家就餐问题的一个最浅显的解法是：为每把叉子单独设一个信号量，哲学家取叉子时执行 wait 操作，放叉子时执行 signal 操作。

```
wait(fork[(i+1)%5]);
signal(fork[(i+1)%5]);
```

这个算法可以保证任何时刻都不会有两个相邻的哲学家同时用餐，但这一解法可能使得每个哲学家都只拿到其左边的叉子，都得不到右边的叉子而陷入死锁。

对于这样的死锁问题有多种避免的方法，以下列出其中的一些方法。

算法 1：至多允许 4 个哲学家同时就餐，以此保证至少有一个哲学家能够就餐，程序描述如下：

```
struct semaphore fork[5] = {1,1,1,1,1};
struct semaphore count = 4;
void philosopher ()
{
    while(1)
    {
        wait(count);                    /* 申请就餐 */
        wait(fork[i]);                  /* 取左边的叉子 */
        wait(fork[(i+1)%5]);            /* 取右边的叉子 */
        eat;
        signal(fork[i]);                /* 放下左边的叉子 */
        signal(fork[(i+1)%5]);          /* 放下右边的叉子 */
        signal(count);                  /* 唤醒其他就餐者 */
        think;
    }
}
void main()
{   cobegin
        Philosopher();
    coend
}
```

算法 2：规定奇数号哲学家先拿左边的叉子，再拿右边的叉子；而偶数号哲学家则相反。按此规定，0、1 号哲学家将竞争 1 号叉子，2、3 号哲学家竞争 3 号叉子。即 5 位哲学家都先竞争奇数号叉子，获得后，再去竞争偶数号叉子，最后总会有一个哲学家能获得两把叉子。

```
struct semaphore fork[5] = {1,1,1,1,1};
void philosopher (int i)                /* i = 0,1,2,3,4 为就餐哲学家的序号 */
{
    while(1)
    {   if(i%2 == 0)                    /* 偶数哲学家 */
```

```
        {
            wait(fork[(i+1)%5]);    /*先取右边的叉子*/
            wait(fork[i]);          /*再取左边的叉子*/
            eat;
            signal(fork[(i+1)%5]);/*放下右边的叉子*/
            signal(fork[i]);        /*放下左边的叉子*/
        }
        else                        /*奇数哲学家*/
        {
            wait(fork[i]);          /*先取左边的叉子*/
            wait(fork[(i+1)%5]);    /*再取右边的叉子*/
            eat;
            signal(fork[(i+1)%5]);/*放下右边的叉子*/
            signal(fork[i]);        /*放下左边的叉子*/
        }
        think;
    }
}
void main()
{   cobegin
        Philosopher(i);
    coend
}
```


观看视频

5.7.3　读者-写者问题

哲学家就餐问题对于多个进程互斥访问有限资源(如I/O设备)这类问题的建模十分有用。另一个著名的同步问题是读者-写者问题,是Courtois等于1971年提出并解决的,它为数据对象(数据文件或记录)的访问建立了一个模型。所谓的读者-写者问题可描述为:有一共享文件L,允许多个进程(读者,Reader)同时读L中的信息,但任何时刻最多只能有一个进程(写者,Writer)写或修改L中的信息。当有进程读时,不允许其他进程写;当有进程写时,不允许其他进程读或写。如何使读者和写者的行为同步呢?

读者-写者问题有两种算法,即读者优先算法和写者优先算法,下面分别论述。

算法1:读者优先。

该算法描述如下:

(1) 当有读进程在读文件时,写进程必须等待。

(2) 当写进程在等待时,后来的读进程可以进入读取。

(3) 只有当没有读进程在读取时,写进程才可写入。

首先,为实现读、写互斥设置一个互斥信号量wmutex,其初值为1。第一个要进入的读者和所有要进入的写者,在对共享文件L操作之前,应执行wait(wmutex)操作;而在退出对L的操作后,应执行signal(wmutex)操作。另外,设置一个整型变量readcount,用以表示某时刻正在读L的进程数目。任一读者在读L前,必须执行readcount加1操作;而在结束读L之后,必须执行readcount减1操作。因此,readcount变量是读者进程都要访问的临界资源,为它设置一个信号量rmutex以保证互斥访问。

读者-写者同步(读者优先)的程序可描述如下:

```
struct semaphore rmutex = 1, wmutex = 1;
```

```
int readcount = 0;
void reader()
{   while(1)
    {   wait(rmutex);                          /* 保证读者间对 readcount 变量的互斥访问 */
        if(readcount == 0) wait(wmutex);       /* 互斥写者 */
        readcount++;
        signal(rmutex);
        perform read operation;
        wait(rmutex);
        readcount -- ;
        if(readcount == 0) signal(wmutex);  /* 开放写操作 */
        signal(rmutex);
    }
}
void writer()
{   while(1)
    {   wait(wmutex);
        perform write operation;
        signal(wmutex);
    }
}
void main()
{   cobegin
        reader(), writer();
    coend
}
```

算法 2：写者优先。

在算法 1 中，当有写者欲写入时，并不能阻止后来的读者进入，因此，写进程可能会处于饥饿状态。下面给出写者优先算法，算法描述如下：

(1) 当有读进程在读时，写进程不得进入。

(2) 当写进程声明欲写入时，后来的读进程不得进入读取。

(3) 只有当所有写进程都离开时，读进程才可进入。

首先，为实现读、写互斥设置两个信号量 read_write 和 write_read，其初值均为 1。所有要进入的读者和第一个要进入的写者，在对共享文件 L 操作之前，应执行 wait(write_read)操作；读者在进入临界区后以及最后一个写者在退出临界区时，应执行 signal(write_read)操作。第一个要进入的读者和所有要进入的写者，在对共享文件 L 操作之前，应执行 wait(read_write)操作；最后一个读者和所有写者在退出临界区时，应执行 signal(read_write)操作。另外，设置两个整型变量 readcount 和 writecount，用以表示某时刻正在读 L 和向 L 写入的进程数目。任一读者在读 L 前，必须执行 readcount 加 1 操作；而在结束读 L 之后，必须执行 readcount 减 1 操作。因此，readcount 变量是读者进程都要访问的临界资源，为它设置一个信号量 rmutex 以保证互斥访问；同理，任一写者在写 L 时，必须执行 writecount 加 1 操作，而在结束写 L 之后，必须执行 writecount 减 1 操作。因此，writecount 变量是写者进程都要访问的临界资源，为它设置一个信号量 wmutex 以保证互斥访问。

读者-写者同步(写者优先)的程序可描述如下：

```
struct semaphore rmutex = 1, wmutex = 1, write_read = 1, read_write = 1;
int readcount = 1, writecount = 1;
```

```
void reader()
{    while(1)
     {    wait(write_read);                    /* 如有写者进入临界区或等待,则阻塞自己 */
          wait(rmutex);                        /* 保证读者间对 readcount 变量的互斥访问 */
          readcount++;
          if(readcount == 1) wait(read_write); /* 互斥写者 */
          signal(rmutex);
          signal(write_read);                  /* 向写者发信号,允许写者阻塞后来的读者 */
          perform read operation;
          wait(rmutex);
          readcount -- ;
          if(readcount == 0) signal(read_write); /* 开放写操作 */
          signal(rmutex);
     }
}
void writer()
{    while(1)
     {    wait(wmutex);                        /* 保证写者间对 writecount 变量的互斥访问 */
          writecount++;
          if(writecount == 1) wait(write_read); /* 互斥读者 */
          signal(wmutex);
          wait(read_write);                    /* 如无读者,则可进行写操作 */
          perform write operation;
          signal(read_write);
          wait(wmutex);
          writecount -- ;
          if(writecount == 0) signal(write_read);      /* 开放读操作 */
          signal(wmutex);
     }
}
void main()
{    cobegin
          reader(), writer();
     coend
}
```

5.8　进程通信


观看视频

进程通信是指进程之间的信息交换。交换的信息量可多可少,少则交换一个状态、数值或少量控制信息,多则需要交换大量的字节。在进程互斥中,通过对信号量的 wait、signal 操作,一个进程可以修改信号量,并向其他进程表明资源是否可用的状态信息。类似地,在生产者-消费者问题中,生产者可借助对信号量的 wait、signal 操作实现与消费者的同步,从而将生产者的消息通过缓冲区传送给消费者。因此,信号量技术可归结为进程通信的一种方式。

信号量机制是一种卓有成效的同步机制,但却不是理想的通信工具。主要表现在以下两方面:

(1) 通信效率低。借助信号量只能在进程之间传递少量信息,如果要交换大量的信息,使用同步机制难以获得令人满意的效果。

（2）通信过程对用户不透明。使用信号量实施进程通信，通信中数据结构的设置及其管理、通信过程中进程的互斥和同步等通信细节问题都必须由通信程序自行负责，这不仅增加了用户编制程序的负担，而且使用不当还会导致通信中的错误，甚至造成死锁。

因此，信号量机制也称为进程的低级通信机制。

为在进程间快速、方便、安全地进行信息交互，应在操作系统中设置一种专门的通信机制，相对于低级通信方式，它应该具有以下特点。

（1）高效性：用户可直接利用通信命令传送大量数据。

（2）透明性：通信细节由操作系统实现并隐藏，当进程的执行依赖一个未到的消息或等待其他进程对信件的答复时，通信机制能自行控制进程的阻塞与唤醒。这样将大大简化程序编制上的复杂性。

本节所要介绍的就是具有这两大通信特点的进程高级通信机制。

随着操作系统的发展，进程通信技术也在不断发展。目前，进程高级通信方式可归结为4大类，即共享存储区系统（Shared-Memory System）、管道（Pipe）通信系统、消息传递系统（Message Passing System）和客户机-服务器系统（Client-Server System）。

5.8.1 共享存储区系统

所谓共享存储区方式，是指在内存中开辟一块共享存储区域作为进程通信区。发送进程把需要交换的信息写入这一区域，而接收进程从该区域中读取信息，以此方式来实现进程间的信息交互。基于共享存储区进行通信的过程可用图5.5表示。共享存储区通信分为建立、附接、读写、断接4个步骤。

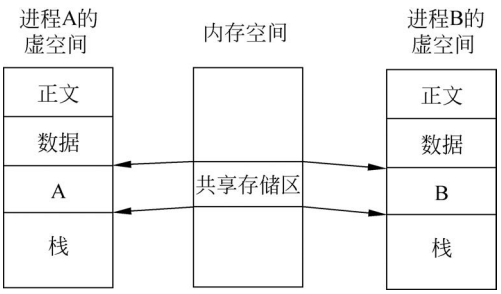

图5.5 利用共享存储区进行通信

1. 建立共享存储区

建立一个共享存储分区，是基于共享存储区通信的前提，通过创建共享存储区的系统调用来完成。该系统调用申请建立关键字为key的共享存储分区。若系统中没有名为key的共享分区，则在共享存储区中创建该分区，并返回该分区的描述符；若该分区已由其他进程建立，则只返回分区描述符。

2. 共享存储区的附接

进程可使用共享存储分区附接的系统调用，将一个共享存储分区附接到自己的虚地址空间上，如图5.5所示，进程A将共享存储分区附接到自己的A区域，进程B将其附接到自己的B区域。利用这一方式，一个共享存储区可以附接到多个进程的虚地址空间上，而同一进程也可附接多个共享存储区。

3. 共享存储区的读写

共享存储区一旦附接到进程的虚地址空间，即成为进程虚地址空间的一部分。进程读写这部分空间，并由内存管理的地址变换机制将这一虚空间映射到指定的共享存储区中，从而实现进程间通信。显然，共享存储区附接到虚地址空间之后，信息的发送与接收就像读、写普通内存一样。

4. 共享存储区的断接

当进程不再需要共享存储区时，可利用共享存储区断接的系统调用断开与指定分区的连接。与共享存储区的连接一旦断开，该存储区便不再对进程有效，不能再作为进程通信的载体。

共享存储区通信的优点在于通信的效率高，常用于对通信速度有较高要求的场合，如UNIX、Linux、Windows 9x 及更高版本、OS/2 等现代操作系统均支持这种通信方式。

5.8.2　管道通信系统

这是在文件系统基础上形成的，利用共享文件实现进程通信的一种方式。所谓管道，是指用于连接多个读写进程，以实现它们之间通信的共享文件。这一文件可被几个进程以不同的使用方式打开。发送者进程以写方式打开管道文件，以字符流的形式将大量数据送入管道，而接收进程则以读方式取得文件中的信息，如图 5.6 所示是管道通信方式的示意图。这种方式首创于 UNIX 系统，现已被引入许多其他操作系统中。

在这一通信方式中，管道文件的建立、打开、读写以及关闭等操作可借用文件系统原有机制来实现。但通信进程在通信过程中的相互协调仅靠文件系统无法解决，需要在文件系统的基础上引入通信协调机制。这里的相互协调有以下三方面的含义。

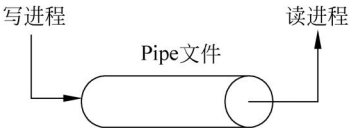

图 5.6　利用管道文件通信

(1) 互斥：进程应互斥地使用通信管道，即当一个进程正在使用管道进行读或写时，其他试图使用管道的进程必须等待。

(2) 同步：读和写进程要相互协调对管道数据的读取速率。当写进程试图向已经写入但尚未读出数据的管道写入时，应阻塞自己，睡眠等待，直至读进程从管道中读取数据后将其唤醒；当读进程试图从空管道中读取数据时，也应睡眠等待，直到写进程将消息写入管道后，再将其唤醒。这样可循环利用有限的管道来传送大量的信息。

(3) 对方是否存在：读和写进程都能以一定方式了解对方是否存在，若写进程了解到读进程已关闭，则不必继续发送信息。

管道通信开销小、交换的信息量大且信息保存期长。但在通信过程中，I/O 操作的次数较多，同步和控制也较为复杂。

5.8.3　消息传递系统

在单机系统、多机系统和网络环境下，进程的高级通信广泛采用消息传递方式。在这种通信方式中，进程间的数据交换以消息(Message)为单位，在计算机网络中，消息也称为报文。发送进程可以直接或间接地将消息传送给目标进程，因此可将这种通信方式分为直接通信和间接通信两种。

1. 直接通信方式

所谓直接通信，是指发送进程利用 OS 所提供的发送命令（原语），直接把消息发送给目标进程。通常，系统提供两条通信原语，分别用于发送和接收消息。

- send(Receiver,message)：表示将消息 message 发送给接收进程 Receiver。
- receive(Sender,message)：表示接收由进程 Sender 发来的消息 message。

基于消息直接通信方式的一个成功实例是消息缓冲通信，由美国的 Hansan 提出，并在 RC4000 系统中实现。后来，这种通信方式被广泛地应用于本地进程之间的通信中。

1）消息缓冲通信机制中的数据结构

在这种通信方式中，操作系统管理着由若干缓冲区构成的通信缓冲池，其中每个缓冲区可放入一个完整的消息，故该缓冲区也称为消息缓冲区，是消息缓冲通信机制中主要利用的数据结构。可描述如下：

```
struct message buffer
{    sender;                          /* 消息发送者进程标识符 */
     size;                            /* 消息长度 */
     text;                            /* 消息正文 */
     next;                            /* 指向下一个消息缓冲区的指针 */
}
```

为完成通信，还需要在 PCB 中增加有关通信的数据项：

```
struct processcontrol block
{    mq;                              /* 消息队列队首指针 */
     mutex;                           /* 消息队列互斥信号量 */
     sm;                              /* 消息队列资源信号量，即消息数目 */
}
```

2）消息发送

为了更有效地利用消息缓冲区的资源，发送进程在发送消息之前，应先在自己的内存空间中设置一个发送区，并在其中组织好欲发送的消息（由发送进程标识符、消息长度和消息正文组成），如图 5.7 所示。

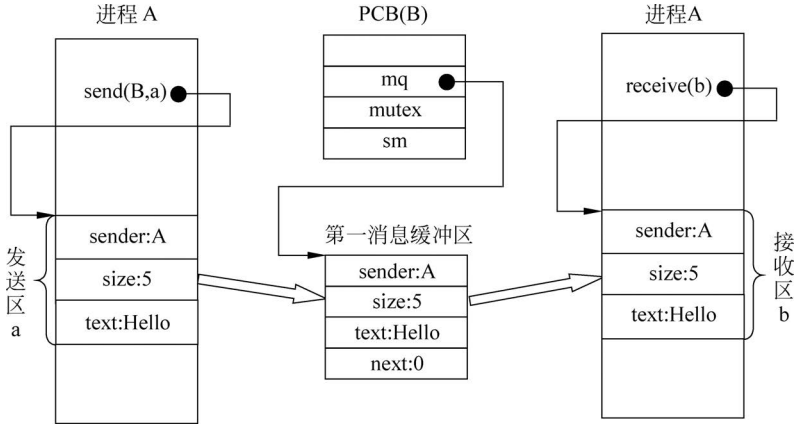

图 5.7　消息缓冲通信

再调用发送原语 send 发送已组织好的消息。其过程是：首先申请一个消息缓冲区，再将发送区中的消息复制到消息缓冲区中，最后将消息缓冲区链接至接收进程消息队列链尾。

发送原语描述如下：

```
void send(receiver,a)
{    getbuf(a.size,i); /* 根据消息长度 a.size 申请一个消息缓冲区 i */
     i.sender = a.sender;
     i.size = a.size;
     i.text = a.text;
     i.next = 0;                          /* 以上是从缓冲区 i 中复制消息 */
     getid(PCB set,reciver,j);            /* 获得接收进程的内部标识符 j */
     wait(j.mutex);                       /* 消息队列是临界资源,必须互斥 */
     insert(j.mq, i);                     /* 将 i 插入消息队列链尾 */
     signal(j.mutex);
     signal(j.sm);                        /* 消息队列长度增 1 */
}
```

3）消息接收

接收进程调用接收原语 receive(b)，获得消息队列 mq 中的第一个消息缓冲区，将其中的消息复制到消息接收区 b 中。接收原语描述如下：

```
void receive(b)
{    j = internal name;                   /* 获得接收进程内部名 */
     wait(j.sm);                          /* 申请一个消息 */
     wait(j.mutex);                       /* 对消息队列互斥操作 */
     remove(j.mq,I)                       /* 从消息队列中移出消息 i */
     signal(j.mutex);
     b.sender = i.sender;
     b.size = i.size;
     b.text = i.text;
}
```

2. 间接通信方式

间接通信方式是指进程之间需要通过某种中间实体来暂存消息。这一中间实体被形象地称为信箱(Mailbox)。当两个进程有一个共享信箱时，它们就能进行通信。一个进程也可以分别与多个进程共享多个不同的信箱，这样，一个进程可以同时和多个进程进行通信。信箱的结构从逻辑上可以分为信箱头和信箱体。信箱头用来存放有关信箱的描述信息，如信箱标识符、信箱的拥有者、信箱口令、信箱的空格数等。信箱体由若干可以存放消息的信箱格组成，信箱格的数目以及每格的大小在创建信箱时确定。信箱中的消息传递可以是单向的，也可以是双向的。图 5.8 为双向信箱通信链路示意图。

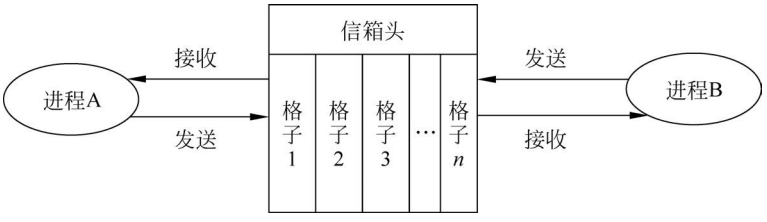

图 5.8　双向信箱通信链路示意图

为支持信箱通信，操作系统提供了若干原语，用于创建和撤销信箱、发送和接收消息等。

1）信箱的创建和撤销

进程可利用信箱创建原语创建一个新信箱，创建者进程是信箱的拥有者，它应该给出所

创建的信箱名以及信箱属性。一个被创建的信箱描述如下：

```
struct mailbox
{    int size;                              /* 信箱大小,即信箱体中的信格数 */
     int count;                             /* 信箱中现有的信件数 */
     struct semaphore S1,S2;                /* S1 为等待信箱信号量,S2 为等待信件信号量 */
     struct semaphore mutex;                /* 信箱互斥访问信号量 */
     mail letter [size];                    /* 信箱体 */
}
```

当进程不再需要信箱时，拥有者可使用撤销原语撤销信箱，但应将这一情况及时通知该信箱的共享者。

2）消息的发送和接收

在间接通信方式中，有发送原语和接收原语两个原语。

- send(struct mailbox B,mail M)：将一封信件 M 发送到信箱 B 指定的位置。
- receive(struct mailbox B,mail M)：从信箱 B 中例行取出第一封信件。

send 原语描述如下：

```
void send(struct mailbox B,mail M)
{    int i;
     wait(B.S1);                            /* 申请信箱中的一个空信格 */
     wait(B.mutex);                         /* 申请信箱的互斥访问权 */
     i = B.count + 1;                       /* 信箱中信件数加 1 */
     B.letter[i] = M;                       /* 把信件放入第 i 个信格 */
     B.count = i;                           /* 设置新的信件数 */
     signal(B.S2);                          /* 信箱中信件数增 1 */
     signal(B.mutex);
}
```

receive 原语描述如下：

```
void receive(struct mailbox B,message X)
{    int i;
     i = 1;
     wait(B.S2);                            /* 申请收信件 */
     wait(B.mutex);                         /* 申请信箱的互斥访问权 */
     B.count = B.count - 1;                 /* 信箱中信件数减 1 */
     X = B.letter[1];                       /* 取信箱中的第 1 封信件 */
     if(B.count!= 0)
     for(i = 1; i <= B.count; i++)
     B.letter[i]: = B.letter[i + 1];        /* 信箱中的所有信件前移 */
     signal(B.S1);                          /* 空信格数加 1 */
     signal(B.mutex);
}
```

利用信箱通信，发送者不必写出接收者，从而可向不知名的进程发送消息，由系统选择接收者。因此，这一通信方式也广泛用于多机系统及计算机网络系统中。它的另一个优点是，消息可以完好地保存在信箱中，只允许核准的目标用户读取。因此，利用信箱通信方式，既可实现实时通信，也可实现非实时通信。

5.8.4 客户机-服务器系统

客户机-服务器系统在当前网络环境下的各种应用领域已成为主流的通信实现机制，它

的主要实现方法有两类：套接字(Socket)和远程过程调用(Remote Procedure Call,RPC)。

1. 套接字

套接字起源于20世纪70年代加州大学伯克利分校版本的UNIX(即BSD UNIX),是UNIX操作系统下的网络通信接口。一开始,套接字被设计用在同一台主机上多个应用程序之间的通信(即进程间的通信),主要是为了解决多对进程同时通信时端口和物理线路的多路复用问题。随着计算机网络技术的发展以及UNIX操作系统的广泛使用,套接字已逐渐成为最流行的网络通信程序接口之一。

一个套接字就是一个通信标识类型的数据结构,包含通信目的地址、通信使用的端口号、通信网络的传输层协议、进程所在的网络地址以及针对客户或服务器程序提供的不同系统调用(或API函数)等,是进程通信和网络通信的基本构件。套接字是为客户机/服务器(C/S模式)设计的,不仅适用于同一台计算机内部的进程通信,也适用于网络环境中不同计算机间的进程通信。由于每个套接字拥有唯一的套接字标识符(也称为套接字号),因此对于来自不同进程或网络连接的通信,都能够加以区分,以确保通信双方的通信链路唯一,便于实现数据传输的并发服务。

常见的套接字工作过程：发送方提供接收方名称,通信双方的进程被分配了一对套接字,一个属于接收进程(或服务器端),另一个属于发送进程(或客户端)。一般情况下,发送进程(或客户端)发出连接请求时,随机申请一个套接字,主机为之分配一个端口,与该套接字绑定,不再分配给其他进程。接收进程(或服务器端)拥有全局公认的套接字和指定的端口(例如FTP服务器监听端口为21,Web或HTTP服务器监听端口为80),并通过监听端口等待客户请求。因此,任何进程都可以向它发出连接请求。接收进程一旦收到请求,就接收来自发送进程的连接,完成连接即可实现进程间的通信。当通信结束时,系统通过关闭接收进程的套接字来撤销连接。

2. 远程过程调用

远程过程(函数)调用是一种通信协议。该协议允许运行于一台主机(本地)系统上的进程调用另一台主机(远程)系统上的进程,而对程序员表现为常规的过程调用,无须额外为此编程。如果涉及的软件采用面向对象编程,那么远程过程调用也可称作远程方法调用。负责处理远程过程调用的进程有两个：本地客户进程和远程服务器进程,这两个进程负责在网络间的消息传递,平时处于阻塞状态,等待消息。

为了使远程过程调用看上去与本地过程调用一样,即希望实现远程过程调用的透明性,使得调用者感觉不到此次调用过程是在其他主机(远程)上执行的,远程过程调用引入了一个存根(Stub)的概念。在本地客户端,每个能独立运行的远程过程都拥有一个客户存根(Client Stub),本地进程调用远程过程实际上是调用该过程关联的存根；与此类似,在每个远程进程所在的服务器端,其所对应的实际可执行进程也存在一个服务器存根(Server Stub)与其关联。本地客户存根与对应的远程服务器存根一般也处于阻塞状态,等待消息。

远程过程调用的主要步骤如下。

(1) 本地过程调用者以一般方式调用远程过程在本地关联的客户存根,传递相应的参数,然后将控制权转移给客户存根。

(2) 客户存根执行,完成包括过程名和调用参数等信息的消息建立,将控制权转移给本地客户进程。

（3）本地客户进程完成与服务器的消息传递，将消息发送到远程服务器进程。

（4）远程服务器进程接收消息后转入执行，并根据其中的远程过程名找到对应的服务器存根，将消息转给该存根。

（5）该服务器存根接到消息后，由阻塞状态转为执行状态，拆开消息并从中取出过程调用的参数，然后以一般方式调用服务器上关联的过程。

（6）在服务器端的远程过程运行完毕后，将结果返回给与之关联的服务器存根。

（7）该服务器存根获得控制权运行，将结果打包为消息，并将控制权转移给远程服务器进程。

（8）远程服务器进程将消息发送回客户端。

（9）本地客户进程接收到消息后，根据其中的过程名将消息存入关联的客户存根，再将控制权转移给客户存根。

（10）客户存根从消息中取出结果，返回给本地调用者进程，并完成控制权的转移。

5.9 典型例题讲解


观看视频

5.9.1 睡眠的理发师问题

【例 5.9】 如图 5.9 所示，一个理发店有等待室和理发室两个房间，等待室有 n 个座位，理发室有一个理发师，理发师同时只能为一位顾客服务。如果没有顾客则理发师睡觉；如果有顾客在等待，则让顾客进入理发室理发。顾客到来时，如果等待室有空座位，则从门 A 进入等待室，如没有空座位则在门外等待。顾客进入等待室后，拉铃通知理发师有顾客到来，如理发师空闲则进入理发室理发，理完发后从门 C 离开理发店，否则在等待室等待。门 A 和门 B 共用一个日本式推拉门，即如果有顾客正从门 A 进入等待室，其他顾客就不能从门 B 进入理发室，反之亦然。试用信号量机制写出诸顾客与理发师之间的同步程序。

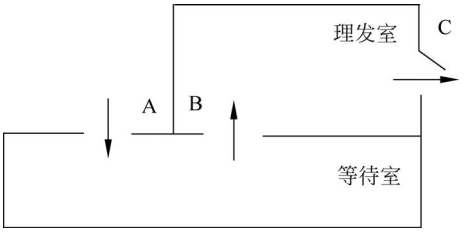

图 5.9 睡眠的理发师问题

【分析】

设置一个互斥信号量 mutex，其初值为 1，控制门 A 和门 B 的互斥使用；设置一个信号量 seat，其初值为 n，其值表示等待室的空座位资源；再设置一个信号量 count，其初值为 0，其值为等待的顾客数；最后设置一个信号量 barber，其初值为 1，表示只有一个理发师资源。

睡眠的理发师问题的程序描述如下：

```
struct semaphore mutex = 1, seat = n, barber = 1, count = 0;
void Customer(void)
{
```

```
    while(1)
    {   wait(seat);                          /* 申请进入等待室 */
        wait(mutex);                         /* 申请使用推拉门 */
        enter the waiting room;              /* 进入等待室 */
        signal(mutex);
        wait(barber);                        /* 申请理发 */
        wait(mutex);                         /* 申请使用推拉门 */
        enter the barbershop;                /* 进入理发室 */
        signal(mutex);
        signal(seat);                        /* 释放座位资源 */
        signal(count);                       /* 顾客数加 1,唤醒理发师 */
        haircut;
        go out;
    }
}
void HairDresser(void)
{
    while(1)
    {
        wait(count);                         /* 申请给顾客理发 */
        cutting…;
        signal(barber);                      /* 通知顾客理发师空闲 */
    }
}
void main()
{   cobegin
        Customer(),HairDresser();
    coend
}
```

5.9.2　银行叫号服务


观看视频

【例 5.10】【2011 年统考真题】某银行提供 1 个服务窗口和 10 个供顾客等待的座位。顾客到达银行时,若有空座位,则到取号机上领取一个号,等待叫号。取号机每次仅允许一位顾客使用。当营业员空闲时,通过叫号选取一位顾客,并为其服务。顾客和营业员的活动过程描述如下:

```
cobegin
{
    process 顾客
    {
        从取号机上获取一个号码;
        等待叫号;
        获取服务;
    }
    process 营业员
    {
        While(1)
        {
            叫号;
            为顾客服务;
        }
    }
}coend
```

请添加必要的信号量和 wait()、signal()操作，实现上述过程中的互斥与同步。要求写出完整的过程，说明信号量的含义并赋初值(本题是 2011 年全国硕士研究生入学考试原题)。

【分析】

(1) 首先要找到本题中的互斥资源：取号机(一次只能有一位顾客领号)，为保证各顾客进程能独立不受干扰地使用该临界资源，因此设一个互斥信号量 mutex。其次，分析本题中需要哪些进程，很明显，需要编写顾客进程和营业员进程。

(2) 本题中的同步问题：顾客需要获得空座位等待叫号，当营业员空闲时，将选取一位顾客并为其服务。空座位的有、无影响等待顾客数量；顾客的有、无决定了营业员是否能开始服务，故分别设置信号量 empty 和 full 来实现这一同步关系。另外，顾客获得空座位后，需要等待叫号和被服务。这样，顾客与营业员之间的服务何时开始又构成了一个同步关系，因此定义信号量 service 来完成这一同步过程。

(3) 最后，从题意中分别设置好各信号量的初始值，根据以上分析，可以编写解答如下：

```
Semaphore empty = 10;                    //空座位的数量
Semaphore mutex = 1;                     //互斥使用取号机
Semaphore full = 0;                      //已占座位的数量
Semaphore service = 0;                   //等待叫号
void Customer()
{   wait(empty);                         //等空位
    wait(mutex);                         //申请使用取号机
    从取号机上取号;
    signal(mutex);                       //取号完毕
    signal(full);                        //通知营业员有新顾客
    wait(service);                       //等待营业员叫号
    接受服务;
    signal(empty);                       //空座位数量 + 1
}
void Salesman()
{   while(1)
    {   wait(full);                      //没有顾客则休息
        signal(service);                 //叫号
        为顾客服务;
    }
}
void main( )
{   cobegin
        Customer( ),Salesman( );
    coend
}
```

本章小结

在多道程序系统中，进程是异步并发的。为保证进程能有序正确地执行，系统必须提供必要的通信机制。通过本章的学习，掌握进程同步与互斥的概念；了解互斥的软件方法及硬件指令机制；重点掌握利用结构型信号量实现进程的同步与互斥，并能写出利用信号量解决实际问题的程序；掌握进程通信的基本方法。

进程同步与互斥的原因是有些进程需要彼此合作，有些进程则需要竞争资源，进程之间

的这些关系如果处理不当，就会产生与时间有关的错误。

临界资源是一次仅允许一个进程所使用的资源，临界区是进程中访问临界资源的那段程序代码。

同步机制是空闲让进、忙则等待、有限等待、让权等待。这是所有同步机制都必须要遵循的准则。

了解同步机制的软件方法和硬件指令机制。

信号量机制是解决进程同步与互斥的有效方法，了解信号量的物理意义，重点掌握利用结构型信号量解决进程的同步与互斥问题。

通过学习几个经典的进程同步问题的解决方法，学会使用信号量机制解决实际问题。

在使用信号量机制解决进程的同步与互斥问题时，首先要仔细分析问题，找出进程间互斥使用临界资源和同步协调的关系，为每一个互斥与同步关系设置相应的信号量，并赋初值。其次要确定有几个进程，其中使用到哪些信号量。用于互斥的信号量的 wait 操作和 signal 操作，在同一个进程的程序段中必须成对出现；而用于同步的信号量的 wait 操作和 signal 操作，一般应在需要同步协作的两个进程中交替出现，即一个进程中进行 wait 操作，另一个进程中进行 signal 操作。最后在编写好进程的临界区后，按照各种可能的进程执行次序进行演练推导，确保进程能够正常运行，不会产生所有进程都被阻塞而无法推进的情况。

信号量是进程间实现通信的低级方法，只能传递少量信息。进程间的高级通信方式有：共享存储区系统、管道通信系统、消息传递系统和客户机-服务器系统。

【本章知识框架】

- 进程通信
  - 进程同步的基本概念：互斥与同步、临界资源与临界区
  - 进程同步机制
    - 软件实现的几种算法(Petersen算法、Dekker算法、Dijkstra算法)
    - 硬件实现的方法
    - 信号量机制：应用信号量实现进程的同步与互斥
    - 管程机制
  - 经典进程同步问题
    - 生产者-消费者问题
    - 哲学家进餐问题
    - 读者-写者问题
  - 进程高级通信方式
    - 共享存储区系统
    - 管道通信系统
    - 消息传递系统
    - 客户机-服务器系统

习题

1. 单项选择题

(1) 有 3 个进程 P_1、P_2、P_3 共享同一个程序段，而每次最多允许两个进程进入该程序段，则信号量 S 的初值为(　　)。

在线答题

A. 0　　B. 1　　C. 2　　D. 3

(2) 用信箱实现进程间互通消息的通信机制要有两个通信原语，它们是(　　)。

A. 就绪原语和接收原语　　B. 发送原语和接收原语

C. 发送原语和执行原语　　D. 就绪原语和执行原语

(3) 下列对临界区的论述中，正确的是(　　)。

A. 临界区是指进程中用于实现进程通信的那段代码

B. 临界区是指进程中用于实现进程互斥的那段代码

C. 临界区是指进程中用于实现进程同步的那段代码

D. 临界区是指进程中用于访问共享资源的那段代码

(4) 若一个信号量的初值为3，经过多次 wait 操作和 signal 操作以后当前值为－1，此表示等待进入临界区的进程数是(　　)。

A. 4　　B. 1　　C. 3　　D. 2

(5) wait 操作可能导致(　　)。

A. 进程就绪　　B. 新进程创建　　C. 进程结束　　D. 进程阻塞

(6) 用 signal 操作唤醒一个等待进程时，被唤醒进程变为(　　)状态。

A. 就绪　　B. 完成　　C. 运行　　D. 等待

(7) 对于两个并发进程，设互斥信号量为 mutex(初值为1)，若 mutex＝0，则(　　)。

A. 表示没有进程进入临界区

B. 表示有一个进程进入临界区

C. 表示有两个进程进入临界区

D. 表示有一个进程进入临界区，另一个进程等待进入

(8) 对信号量 S 执行 wait 操作后，使进程进入等待队列的条件是(　　)。

A. S.value＜0　　B. S.value＞＝0　　C. S.value＜＝0　　D. S.value＞0

(9) 在操作系统中，要对并发进程进行同步的原因是(　　)。

A. 进程具有动态性　　B. 进程必须在有限的时间内完成

C. 进程具有结构性　　D. 并发进程是异步的

(10) 有三个进程共享同一程序段，而每次只允许两个进程进入该程序段，若用 wait 操作和 signal 操作同步机制，则信号量 S 的取值范围是(　　)。

A. 1，0，－1，－2　　B. 3，2，1，0

C. 2，1，0，－1　　D. 2，1，0，－1，－2

(11)【2010年考研真题】设与某资源关联的信号量初值为3，当前值为1。若 M 表示该资源的可用个数，N 表示等待该资源的进程数，则 M、N 分别是(　　)。

A. 2、0　　B. 0、1　　C. 1、2　　D. 1、0

(12)【2011年考研真题】有两个并发执行的进程 P_1 和 P_2，共享初值为1的变量 x。P_1 对 x 加1，P_2 对 x 减1。加1和减1操作的指令序列分别如下所示。

```
//加 1 操作                          //减 1 操作
load R1,x      //取 x 到寄存器 R1 中   load R2,x      //取 x 到寄存器 R2 中
inc R1                               dec R2
store x,R1     //将 R1 的内容存入 x    store x,R2     //将 R2 的内容存入 x
```

两个操作完成后，x 的值(　　)。

A. 可能为－1 或 3　　B. 只能为 1

C. 可能为 0、1 或 2　　D. 可能为－1、0、1 或 2

(13)【2014 年考研真题】下列关于管道(Pipe)通信的叙述中，正确的是(　　)。

A. 一个管道可实现双向数据传输

B. 管道的容量仅受磁盘容量大小限制

C. 进程对管道进行读操作和写操作都可能被阻塞

D. 一个管道只能有一个读进程或一个写进程对其操作

(14)【2016 年考研真题】下列关于管程的叙述中，错误的是(　　)。

A. 管程只能用于实现进程的互斥

B. 管程是由编程语言支持的进程同步机制

C. 任何时候只能有一个进程在管程中执行

D. 管程中定义的变量只能被管程内的过程访问

(15)【2018 年考研真题】若 x 是管程内的条件变量，则当进程执行 x.wait() 时所做的工作是(　　)。

A. 实现对变量 x 的互斥访问

B. 唤醒一个在 x 上阻塞的进程

C. 根据 x 的值判断该进程是否进入阻塞状态

D. 阻塞该进程，并将之插入 x 的阻塞队列中

(16)【2018 年考研真题】在下列同步机制中，可以实现让权等待的是(　　)。

A. Peterson 方法　　B. swap 指令

C. 信号量方法　　D. Test-and-Set 指令

(17)【2020 年考研真题】下列准则中，实现临界区互斥机制必须遵循的是(　　)。

Ⅰ. 两个进程不能同时进入临界区

Ⅱ. 允许进程访问空闲的临界资源

Ⅲ. 进程等待进入临界区的时间是有限的

Ⅳ. 不能进入临界区的执行态进程立即放弃 CPU

A. Ⅰ，Ⅳ　　B. Ⅱ，Ⅲ　　C. Ⅰ，Ⅱ，Ⅲ　　D. Ⅰ，Ⅲ，Ⅳ

2. 填空题

(1) 临界资源是_____的资源，临界区是_____。

(2) 当信号量 $s>0$ 时，表示_____；当 $s=0$ 时，表示_____；当 $s<0$ 时，表示_____。

(3) 设计进程同步机制的准则有_____、_____、_____和_____。

3. 基本概念解释和辨析

(1) 同步与互斥。

(2) 临界资源与临界区。

(3) 高级通信与低级通信。

(4) 直接通信与间接通信。

4. 论述题

（1）什么是“忙等待”？如何克服“忙等待”？

（2）在解决进程互斥时，如果 TS 指令的执行可以中断，会出现什么情况？而如果 wait、signal 操作的执行可分割，又会出现什么情况？

（3）设有 n 个进程共享一互斥段，对于如下两种情况：

① 每次只允许一个进程进入临界区。

② 最多允许 m 个进程（$m<n$）同时进入临界区。

可以采用怎样的信号量？信号量值的变化范围如何？

（4）下面是两个并发执行的进程，它们能正确执行吗？若不能正确执行，请举例说明，并改正（x 是公共变量）。

```
void P1()
{    int y, z;
     x = 1;
     y = 0;
     if(x >= 1) y = y + 1;
     z = y;
}
void P2()
{    int t, u;
     x = 0;
     t = 0;
     if(x < 1) t = t + u;
     u = t;
}

void main()
{    cobegin
          P1(),P2();
     coend
}
```

（5）共享存储区通信是如何实现的？

（6）假设某系统未直接提供信号量机制，但提供了进程通信工具。如果某程序希望使用关于信号量的 wait()、signal()操作，那么该程序应如何利用通信工具模拟信号量机制？要求说明如何用 send()、receive()操作及消息表示 wait()、signal()操作及信号量。

5. 应用题

（1）有 3 个并发进程 R、W_1 和 W_2，共享两个各可存放一个数字的缓冲区 B_1、B_2。进程 R 每次从输入设备读入一个数字，若读入的是奇数，则将它存入 B_1 中，若读入的是偶数，将它存入 B_2 中。若 B_1 中有数，则由进程 W_1 将其打印输出；若 B_2 中有数，则由进程 W_2 将其打印输出。试编写保证三者正确工作的程序。

（2）桌上有一空盘，可放一个水果。爸爸可向盘中放苹果，也可向盘中放橘子，儿子专等吃盘中的橘子，女儿专等吃盘中的苹果。规定一次只能放一种水果，试写出爸爸、儿子、女儿正确同步的程序。

（3）放小球问题：一个箱子里只有白色和黑色两种小球，且数量足够多。现在需要从中取出一些小球放入一个袋子中。约定：①一次只能放入一个小球；②白球的数量至多只能比黑球少 N 个，至多只能比黑球多 M 个（M、N 为正整数）。请用信号量机制实现进程的

同步与互斥。

(4)【2009年考研真题】3个进程P_1、P_2、P_3互斥使用一个包含$N(N>0)$个单元的缓冲区。P_1每次用produce()生成一个正整数并用put()送入缓冲区某一空单元中，P_2每次用getodd()从该缓冲区中取出一个奇数并用countodd()统计奇数个数，P_3每次用geteven()从该缓冲区中取出一个偶数并用counteven()统计偶数个数。请用信号量机制实现这3个进程的同步与互斥活动，并说明所定义的信号量的含义。要求用伪代码描述。

(5)【2013年考研真题】某博物馆最多可容纳500人同时参观，博物馆只有一个出入口，该出入口一次仅允许一人通过，参观者的活动描述如下：

```
cobegin
     参观者进程 i:
    {
        …
        进门
        …
        参观
        …
        出门
        …
    }
coend
```

请添加必要的信号量和wait、signal操作，以实现上述进程的互斥和同步，要求写出完整的过程，说明信号量的含义并赋初值。

(6)【2014年考研真题】系统中有多个生产者进程和多个消费者进程，共享一个能存放1000件产品的环形缓冲区(初始为空)。当缓冲区未满时，生产者进程可以放入其生产的一件产品，否则等待；当缓冲区未空时，消费者进程可以从缓冲区取走一件产品，否则等待。要求一个消费者进程从缓冲区连续取出10件产品后，其他消费者进程才可以取产品。请使用信号量P、V(或wait()、signal())操作实现进程间的互斥与同步，要求写出完整的过程，并说明所用信号量的含义和初值。

(7)【2015年考研真题】有A、B两人通过信箱进行辩论，每个人都从自己的信箱中取得对方的问题。将答案和向对方提出的新问题组成一个邮件放入对方的邮箱中。假设A的信箱最多放M个邮件，B的信箱最多放N个邮件。初始时A的信箱中有x个邮件$(0<x<M)$，B的信箱中有y个邮件$(0<y<N)$。辩论者每取出一个邮件，邮件数减1。A和B两人的操作过程描述如下：

```
CoBegin
```

A{
while(TRUE)
{
　　从A的信箱中取出一个邮件;
　　回答问题并提出一个新问题;
　　将新邮件放入B的信箱;
}
}
	B{
	while(TRUE)
	{
	　　从B的信箱中取出一个邮件;
	　　回答问题并提出一个新问题;
	　　将新邮件放入A的信箱;
	}
	}

```
CoEnd
```

当信箱不为空时，辩论者才能从信箱中取邮件，否则等待。当信箱不满时，辩论者才能将新邮件放入信箱，否则等待。请添加必要的信号量和 P、V（或 wait()、signal()）操作，以实现上述过程的同步。要求写出完整过程，并说明信号量的含义和初值。

(8)【2019 年考研真题】有 $n(n\geqslant 3)$ 位哲学家坐在一张圆桌边，每位哲学家交替地就餐和思考。在圆桌中心有 $m(m\geqslant 1)$ 个碗，每两位哲学家之间有 1 根筷子。每位哲学家必须取到一个碗和两侧的筷子之后，才能就餐，进餐完毕，将碗和筷子放回原位，并继续思考。为使尽可能多的哲学家同时就餐，且防止出现死锁现象，请使用信号量的 P、V 操作（wait()、signal()操作）描述上述过程中的互斥与向步，并说明所用信号量及初值的含义。

(9)【2020 年考研真题】现有 5 个操作 A、B、C、D 和 E。它们满足下列条件：操作 C 必须在 A 和 B 完成后执行，操作 E 必须在 C 和 D 完成后执行，请使用信号量的 wait()、signal()操作描述上述同步关系，并说明所用信号量及其初值。

(10)【2022 年考研真题】某进程的两个线程 T_1 和 T_2 并发执行 A、B、C、D、E 和 F 共 6 个操作，其中 T_1 执行 A、E 和 F，T_2 执行 B、C 和 D。图 5.10 表示上述 6 个操作的执行顺序所必须满足的约束：C 在 A 和 B 完成后执行，D 和 E 在 C 完成后执行，F 在 E 完成后执行。请使用信号量的 wait()、signal()操作描述 T_1 和 T_2 之间的同步关系，并说明所用信号量的作用及其初值。

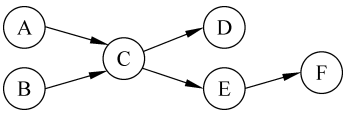

图 5.10　必须满足的约束