第 3 章　进程同步与进程间通信

多个进程并发执行时，有些进程是相互独立的，有些进程之间是相互影响、相互合作的。相互影响、相互合作的进程之间存在制约关系，从而引发了进程同步与进程间通信。进程同步与进程间通信是单用户、单任务操作系统中不曾出现的，若处理不好，会影响进程的执行过程，甚至产生错误的运行结果，所以操作系统设计者必须对该问题给予高度重视。进程同步与进程间通信是本书的重点和难点之一，也是各种考试中频繁考查的知识点。因此，本章适当增加了例题数量，用于更好地掌握进程的同步与通信。

通过本章的学习，要学会用并发执行的思维方式分析和看待操作系统中的并发执行进程，时刻注意多个并发执行进程同步和进程间通信问题。

【本章学习目标】

- 进程同步和进程互斥以及两者之间的区别。
- 临界资源和临界区。
- 整型信号量、记录型信号量的代码定义。
- 信号量及其在实际问题中的应用。
- 管程的定义和使用。
- 进程之间通信的几种方式。

3.1　进程同步和进程互斥

3.1.1　进程同步和进程互斥的基本概念

并发执行的多个进程在异步环境中运行时，每个进程都以各自独立的、不可预知的速度向前推进。若各进程之间不存在制约关系，则可任意并发执行。在实际中，由于多个进程合作完成用户任务或竞争共享资源，所以进程之间存在相互制约关系。相互合作的进程需要在某些特定执行点上协调它们的工作，从而限制进程的彼此执行速度，称为进程之间的直接制约关系。例如，计算进程和打印进程并发执行，打印进程必须等待计算进程得出计算结果后，才能进行打印输出；计算进程要求打印进程将上一次计算的结果打印输出后，才能进行下一次计算。由于一些进程共享某个资源，而这类资源必须互斥使用，从而导致进程之间存在相互制约关系。这种关系称为进程之间的间接制约关系。例如，若干并发进程共享同一台打印机，若在某个进程正在使用打印机时其他进程也想使用该打印机，则必须等待；只有在该进程放弃使用该打印机后，其他进程才能使用。这几个并发执行进程之间存在着间接制约关系。

在多道程序环境下，操作系统必须采取措施处理好进程之间的制约关系。进程同步的主要任务是对多个有制约关系的进程的执行次序进行协调，以使并发进程之间能有效地、安全地互相合作和共享系统资源。

1. 进程同步与进程互斥

并发执行的进程因直接制约关系而需相互等待、相互合作，以实现各进程按相互协调的速度向前推进，这种协调过程称为进程同步。因间接制约而导致进程交替执行的过程称为进程互斥。从广义上讲，进程互斥可看作进程同步的特例，在某些参考文献中把进程同步和进程互斥统称为进程同步。本书为了方便讲解，仍沿用进程同步和进程互斥的说法。如用户处理不好进程之间的同步和互斥关系，会造成进程执行结果不正确，甚至出现进程死锁现象。进程同步与进程互斥的比较如表 3.1 所示。

表 3.1　进程同步与进程互斥的比较

比较项	进 程 同 步	进 程 互 斥
区别	进程与进程之间有序合作。 相互清楚对方的存在及其作用，直接合作。 多个进程合作完成一个任务。 例如，发送消息进程和接收消息进程之间；输入进程、计算进程和输出进程之间等	进程与进程之间共享临界资源。 不清楚对方的情况，只是共享同一个临界资源。 各个进程之间没有任何合作工作。 例如，共享打印机的若干进程之间；共享同一全局变量的若干进程之间等

唯物辩证法认为，世界上的任何事物都不是孤立存在的，彼此之间存在相互影响、相互制约、相互作用的关系。多个并发执行的进程之间也是如此，存在大量、复杂的同步和互斥关系。

2. 临界资源与临界区

临界资源又称独占资源或互斥资源，是指某段时间内只允许一个进程访问的资源。例如，打印机等硬件资源，以及只能互斥使用的变量、表格、队列等软件资源。临界资源的使用只能采用互斥方式。当一个进程正在使用某个临界资源且尚未使用完毕时，想使用该资源的其他进程就必须阻塞等待。只有当进程释放该资源后，其他进程才可被唤醒并申请使用该资源。任何进程不能在其他进程没有使用完临界资源时使用该资源，否则将会造成混乱，导致错误。因此，对临界资源必须进行保护，避免两个或多个进程非互斥方式访问这类资源。

例题 3.1　进程 A 和进程 B 共享一个计数变量 counter，进程 A 对它做加 1 操作，进程 B 对它做减 1 操作，这两个操作如用机器语言实现，常采用如下所示的形式。

进程 A	进程 B
A_1. 将 counter 读入通用寄存器 R_1 中	B_1. 将 counter 读入通用寄存器 R_2 中
A_2. R_1 的值做加 1 操作	B_2. R_2 的值做减 1 操作
A_3. 将 R_1 的值写回变量 counter 中	B_3. 将 R_2 的值写回变量 counter 中

设当前 counter＝5，则不同的执行顺序就会产生不同执行结果，如下：

A_1,A_2,A_3,B_1,B_2,B_3　　counter＝5

B_1,B_2,B_3,A_1,A_2,A_3　　counter＝5

B_1,B_2,A_1,A_2,A_3,B_3　　counter＝4

A_1,A_2,B_1,B_2,B_3,A_3　　counter＝6

这表明程序的执行已失去再现性。在后两个执行顺序上，由于进程 B 使用 counter 时，进程 A 插入并使用了 counter 或者进程 A 使用 counter 时，进程 B 插入并使用了 counter，

造成结果错误。细心的读者会发现，造成计数变量 counter 不正确的原因与进程 A、B 被打断的时间有关，由这种原因造成的结果错误常被称为与时间有关的错误。

为了预防此类错误，关键是将 counter 作为临界资源处理，让进程 A、B 互相排斥地访问变量 counter。

各个进程中访问临界资源的、必须互斥执行的那部分程序段被称为临界区，如图 3.1 所示。

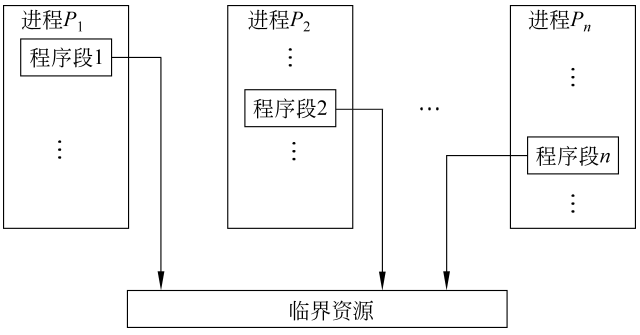

图 3.1　多个并发执行进程的临界区

图 3.1 中的程序段 1、2、…、n 是分布于不同程序中但访问同一临界资源的临界区，它们在各自程序中的位置不同，但它们之间必须互斥执行。由于临界资源必须互斥使用，进程在进入其临界区、执行临界区代码前，首先要检查是否有其他进程正在使用该临界资源，执行检查工作的这段代码称为进入区。如果有其他进程正在使用该临界资源（即正在执行进程内部关于该临界资源的临界区代码），则该进程必须等待；如果没有，则该进程才能进入临界区，执行临界区代码，同时关闭临界区，以防其他进程再次进入该临界区。当进程用完临界资源退出临界区时，要将临界区访问标识置为未访问，以便其他进程能够进入该临界区，这段代码称为退出区。在退出区代码中，进程需通知其他等待使用该临界资源的进程可以使用该临界资源了，并把其从阻塞状态唤醒为就绪状态。

使用临界资源的代码结构如图 3.2 所示。

临界区以前的代码

进入临界区代码-进入区

临界区代码

退出临界区代码-退出区

临界区以后的其余部分代码

图 3.2　使用临界资源的代码结构

有了临界资源和临界区的概念，进程间的互斥可被描述如下：禁止两个或两个以上的进程同时进入各自进程中的、访问同一临界资源的临界区。

上例中，进程 A 和进程 B 对计数值 counter 进行修改的语句 A_1、A_2、A_3 和 B_1、B_2、B_3 都应该划入临界区。通过临界区保护机制保证一个进程在临界区执行时，不让另一个进程进入相关的临界区执行，就不会造成与时间有关的错误。这样一来，并发执行的进程 A 和进程 B 只有两个推进次序，即 A_1、A_2、A_3、B_1、B_2、B_3 和 B_1、B_2、B_3、A_1、A_2、A_3，这两个推

进次序的运算结果均为5,都是正确的。因此,合理设定临界区并实现互斥访问机制,是保证并发程序正确性的重要手段。

3. 同步机制应遵循的准则

为防止两个进程同时进入临界区,可采用软件方法或同步机构进行协调。不论是软件方法还是同步机构都应遵循以下准则。

(1) 空闲让进。若没有进程处于某临界资源对应的临界区,则表明该临界资源处于空闲状态,应允许一个请求进入该资源临界区的进程立即进入临界区。

(2) 忙则等待。若有进程已进入某临界资源对应的临界区,则表明该临界资源正在被使用,因而其他试图进入该临界资源对应临界区的进程必须在进入区代码处等待。

(3) 有限等待。对要求访问临界资源的进程应保证其在有限时间内能进入自己的临界区,以免陷入"死等"状态。

(4) 让权等待。当进程不能进入自己的临界区时,应立即阻塞自己并释放处理机,以免进程陷入"忙等"状态。

4. 实现临界区互斥的基本方法

如何按照同步机制应遵循的准则实现对临界区的管理呢? 通常有软件和硬件两种方法。

1) 软件实现方法

软件实现方法是指编程人员编写程序时在临界区前面设置检查语句,如果有其他并发执行的进程在临界区中,则不允许进程进入临界区,只能在临界区外"忙等"或阻塞。只有当其他进程退出临界区后,进程才能够进入临界区运行。

常见的软件实现方法有 Dekker 算法和 Peterson 算法等。

(1) Dekker 算法。有两个进程 P_i、P_j,设立一个共享全局整型变量 turn,用于描述允许进入临界区的进程标识。在进入区中循环检查是否允许本进程进入:turn 为 i 时,进程 P_i 可进入;在退出区中修改允许进入的进程标识:当进程 P_i 退出时,将 turn 改为进程 P_j 的标识 j。

设立一个布尔型的标识数组 flag[2],用于描述进程是否在临界区中,初值均为 false。标识数组是进程进入临界区的钥匙,使用时进程先修改自己的标识,然后检查另一个进程的标识。通常每个进程设置自己的 flag,表明自己想进入临界区,但也可根据情况重置自己的 flag,以尊重另一个进程:

```
boolean flag[2];
int turn;
void P0()
{
    while (true)
    {
        flag[0]=true;                    //进入区
        while (flag[1])
        {
              if (turn==1)
              {
                  flag[0]=false;
```

```
                while (turn==1)
                    {/*  什么也不做  */}
                flag[0]=true;
            }
        临界区代码;
        turn=1;                          //退出区
        flag[0]=false;
        进程的其余部分;
        }
    }
}
void P1()
{
    while (true)
    {
        flag[1]=true;                    //进入区
        while (flag[0])
        {
            if (turn==0)
            {
                flag[1]=false;
                while (turn==0)
                    {/*  什么也不做  */}
                flag[1]=true;
            }
        临界区代码;
        turn=0;                          //退出区
        flag[1]=false;
        进程的其余部分;
        }
    }
}
void main()
{
    turn=0;
    flag[0]=false;
    flag[1]=false;
    parbegin(P0(),P1());
}
```

当 P_0 希望进入自己的临界区时，它把自己的 flag 置为 true，然后继续检查 P_1 的 flag。如果 P_1 的 flag 为 false，P_0 可立即进入自己的临界区；如 P_1 的 flag 为 true，P_0 检查 turn，如果它发现 turn＝1，则将自己的 flag 置为 false，并一直等到 turn＝0 后再将自己的 flag 置成 true，进入临界区。退出临界区时，P_0 把自己的 flag 置为 false，并把 turn 置为 1，表示 P_1 可以进入。

Dekker 算法解决了互斥问题，但比较复杂。Peterson 提出了一种较简单的算法。

(2) Peterson 算法。标识数组 flag 标识每个进程是否在临界区。整型变量 turn 解决

同时进入时的冲突问题，turn＝j 时表示可进入的进程为 j。想进入临界区的进程在进入区中先修改自己的 flag 为 true，并修改 turn。然后，检查对方的 flag，如果不在临界区则自己进入。否则，再检查 turn，turn 保存的是较晚的一次赋值，较晚进程等待。

```
boolean flag[2]={false,false};
int turn;
void P0()
{
    while (true)
    {
        flag[0]=true;                    //进入区
        turn=1;
        while (flag[1]&& turn==1)
        {/*什么也不做*/;}
        临界区代码;
        flag[0]=false;                   //退出区
        进程其余部分;
    }
}
void P1()
{
    while (true)
    {
        flag[1]=true;                    //进入区
        turn=0;
        while (flag[0]&&turn==0)
        {/*什么也不做*/;}
        临界区代码;
        flag[1]=false;                   //退出区
        进程其余部分;
    }
}
```

下面以进程 P_0 想进入临界区为例，说明该算法的执行过程。先设置 flag[0]＝true，turn＝1；然后检查 flag[1]，如果 flag[1]为 false，则直接进入临界区；否则，此时 P_0、P_1 均要求进入临界区，产生冲突。再检查 turn，由于 turn 保存的是较晚的一次赋值，如果 turn 为 1，则表示 P_0 为后要求进入临界区的，因此 P_0 等待。当进程离开临界区后，将 flag[0]设为 false。

其实该算法可以类比为下列情况：有两个人进门，每个人进门前都会和对方客套一句“你先走”。如果进门时没别人，就当和空气说句废话，然后大步登堂入室。如果两个人同时进门，就相互请先，但各自只客套一次，所以先客套的人请完对方，就等着对方请自己，然后光明正大地进门。

2）硬件实现方法

（1）中断禁用。为保证多个并发进程互斥使用临界资源，只需保证一个进程在执行临界区代码时不被中断即可，这可通过操作系统内核专为启用和禁用中断而定义的原语来

实现。

进程可通过下面的方法实现互斥。

```
while (1)
{
    禁止中断;
    临界区;
    启用中断;
    进程其余部分;
}
```

由于临界区不能被中断,所以可实现互斥。该方法代价太高,这是因为若进程在临界区中产生中断后,CPU 只能忙等,系统执行效率明显降低。这种方法仅适用于单处理器系统且临界区较短的情况。

(2) 专用机器指令。编程人员利用专用机器指令实现临界区的互斥使用。在临界区代码前通过硬件指令来检查某一全局变量是否有其他并发执行的进程在临界区中使用。若没有,则可进入临界区;若有,则重复检查,进程处于“忙等”状态。当进程执行完临界区代码退出时,修改该全局变量,允许其他并发执行的进程进入临界区执行。

通常机器指令都是原语操作,在指令执行期间不允许被中断。常见的硬件实现指令有 TestAndSet、Swap 等指令。

TestAndSet 指令定义如下:

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

该指令测试参数 i 的值。如果 i 为 0,则用 1 取代并返回 true,这表示临界资源未被使用,进程可占用临界资源;如果 i 为 1,则 i 值不变,返回 false。这表示临界资源已被使用,进程暂时不能占用临界资源。由于整个 TestAndSet 函数自动整体执行,可屏蔽任何中断,故可实现进程互斥。

用 TestAndSet 指令实现互斥举例如下:

```
const int n=N;              /*N为进程数*/
int lock;
void P(int i)
{
    while (1)
    {
        while (!TestAndSet (lock))
```

```
            /*什么也不做*/;
        临界区
        lock=0;
            /*其余部分*/
    }
}
void main()
{
    lock=0;
    parbegin (P(1),P(2),…,P(n));
}
```

n 个并发执行进程都在循环检测 lock 变量。只有当 lock 为 0 时,才允许其中一个进程进入临界区,实现对临界资源的互斥访问;lock 不为 0 时,各进程一直在临界区外循环检测。

Swap 指令定义如下:

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

该指令交换一个寄存器和一个存储单元的内容。在该指令的执行过程中,其他任何指令对该存储器单元的访问均被阻止。

用 Swap 指令实现互斥举例如下:

```
const int n=N;              /*N为进程数*/
int lock;
void P(int i)
{
    int key;
    while ( 1 )
    {
        key=1;
        while (key!=0)
            Swap (key,lock);
        临界区
        Swap (key,lock);
        /*进程其余部分*/
    }
}
void main()
{
    lock=0;
    parbegin (P(1),P(2),…,P(n));
}
```

共享变量 lock 被初始化为 0，每个进程都有一个局部变量 key 且初始化为 1。唯一可进入临界区的进程是发现 lock 等于 0 的那个进程，它把 lock 置为 1，防止其他进程进入临界区。该进程离开临界区时，把 lock 重置为 0，允许其他进程进入临界区。

上述硬件指令能简单、有效地实现进程互斥，适用于运行在单 CPU 或共享主存的多 CPU 上的任何数目的并发进程间。但当临界资源已被占用时，其他欲访问临界资源的进程必须不断地进行测试，处于“忙等”状态，造成处理机资源的极大浪费。此外，进程也很容易进入“死等”状态，因为当一个进程离开一个临界区并且有多个进程等待时，系统选择哪个等待进程是随意的，这会造成某些进程长时间等待，出现“死等”现象。因此，很难将上述方法应用于解决复杂的进程间同步场景。

实现临界区互斥的软件和硬件方法都相对简单，虽然能实现临界区的互斥，但使用效率较差，用户在编程时要时刻关注临界资源的互斥逻辑是否实现，从而增加了用户的编程负担。

3.1.2 信号量机制

信号量(semaphore)机制是一种高效的进程同步机制。1965 年，荷兰著名计算机科学家 Edsger W. Dijkstra 在其有关协作式顺序处理的信号量论文中首先提出信号量的概念。信号量除了能实现对临界区的互斥访问，还能实现进程之间复杂的同步关系。

信号量的基本思想如下：两个或多个进程可以利用彼此之间收发的简单信号实现“正确的”并发执行，一个进程在收到一个指定信号前，会被迫在一个指定位置暂停下来，从而实现进程间的同步或互斥。

信号量包含一个受保护的整型变量。和一般整型变量不同，该整型变量的值在初始化后，只能由两个原语(*P* 原语和 *V* 原语)之一访问和修改。因此，信号量通常只能进行 3 个操作：初始化、*P* 原语和 *V* 原语。

注意：*P* 是荷兰语 proberen 的首字母，意为“to test”，*V* 是荷兰语 verhogen 的首字母，意为“to increment”。有的教材或习题集中称 *P* 原语为 wait 原语，称 *V* 原语为 signal 原语。

常见信号量分为以下两类。

(1) 整型信号量。整型信号量是一个表示资源数目的整型变量 *S*。它和其他普通整型变量不同，只能对其进行以下 3 种操作。

① 初始化操作。例如，*S*＝1；*S* 的值表示系统中某类可用资源的数量。

② *P* 原语操作。例如，*P*(*S*)或 wait(*S*)。

P 原语的伪代码如下：

```
P(S):
    while S≤0 do no-op
    S=S-1;
```

其中，no-op 为不进行任何操作，返回循环判定条件，进行下一轮循环的判定。

③ *V* 原语操作。例如，*V*(*S*)，或 signal(*S*)。

V 原语的伪代码如下：

```
V(S): S=S+1;
```

$P(S)$操作和$V(S)$操作是两个原子操作，执行期间不能被中断。

注意：整型信号量的 P 操作中，只要是信号量 $S\leqslant 0$，就会不断地测试。因此，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。这降低了处理器的使用效率。

(2) 记录型信号量。整型信号量机制中的 P 操作，只要信号量 $S\leqslant 0$，就会不断地循环判定，进行“忙等”，该机制违反了“让权等待”的准则。记录型信号量就是为解决这种“忙等”现象而被提出来的。

记录型信号量的数据结构可用一个结构体表示如下：

```
type semaphore=record
    value: integer;
    L: list of process;          //阻塞进程队列
end.
```

记录型信号量机制中除了一个用于代表资源数目的整型变量 value 外，还增加了一个进程链表指针 L，它指向在该信号量上阻塞的进程队列。该队列按阻塞的先后顺序记录了由于不能访问同一临界资源而阻塞的各个进程。在以后的章节中，都默认采用记录型信号量。

相应地，$P(S)$和 $V(S)$操作可描述如下：

```
procedure P(S)
var S:semaphore;
begin
    S.value:=s.value-1;
    if S.value<0 then block(S.L)
end.
procedure V(S)
var S:semaphore;
begin
    S.value:=s.value+1;
    if S.value≤0 then wakeup(S.L)
end.
```

在记录型信号量中，P 操作仍然对信号量的值减 1。当 S.value<0 时，调用阻塞原语 block($S.L$)使执行该 P 原语的进程阻塞，并插入到链表 $S.L$ 中。由于阻塞进程会主动释放 CPU 给其他进程使用，这有效地避免了“忙等”。此时，S.value 的绝对值表示在该信号量链表中阻塞的进程数目。

对信号量执行一次 V 操作，表示执行进程释放了一个单位资源，系统中可供分配的该类资源数目 S.value 加 1。加 1 后，如果 S.value$\leqslant 0$ 时，说明在该信号量上还有其他阻塞的进程，需调用唤醒原语 wakeup($S.L$)唤醒一个在该信号量上阻塞的其他进程，使其状态由阻塞状态转换为就绪状态，等待调度执行。

3.1.3 利用信号量解决进程互斥问题

为实现多个进程互斥地访问临界资源，只需为该资源设置一个公有的互斥信号量mutex，设其初值为1，表明该临界资源未被占用。然后，将进程中使用该临界资源的临界区置于 P(mutex)和 V(mutex)操作之间，即可实现多个进程对临界资源的互斥访问。下面，以两个进程为例说明其实现过程：

```
semaphore mutex=1;
进程 P1:                    进程 P2:
P(mutex);                   P(mutex);
CS1;     //临界区           CS2;     //临界区
V(mutex);                   V(mutex);
```

每个欲访问临界资源的进程，在进入临界区之前，都要对公有信号量 mutex 进行 P 操作。如果此刻临界资源未被访问，即 mutex=1，则此次 P 操作成功，mutex 被置为 0，随后，进程进入临界区执行。而此时如果其他进程也想进入自己的访问该资源的临界区，其 P 操作由于 mutex 为 0 而失败，从而被阻塞。这就保证了进程对该临界资源的互斥访问。当占用该临界资源的进程使用完毕退出临界区时，执行 V 操作，释放该资源，将 mutex 加 1。若 mutex≤0，则说明在该信号量上还有阻塞进程，用唤醒原语 wakeup 唤醒其中一个阻塞进程。

上述情况可以推广到多个进程访问同一临界资源，P、V 操作解决多个进程互斥问题的方法如图 3.3 所示。

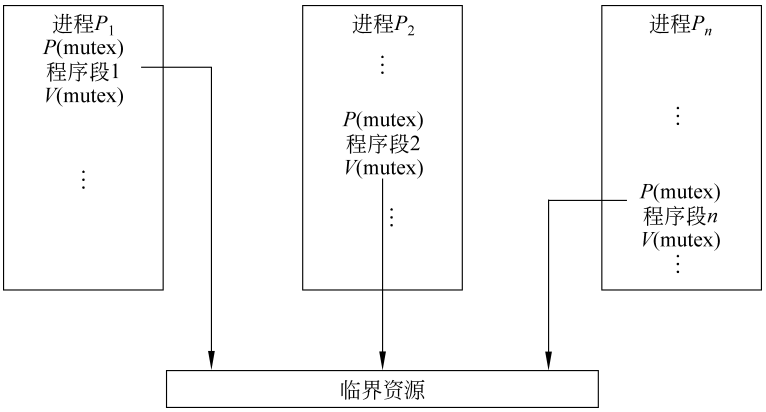

图 3.3 用信号量及 P、V 操作解决进程间互斥问题

利用信号量实现进程互斥时，同一个进程中的 P 和 V 操作应成对出现，且使用该临界资源的临界区应放在两个操作之间。如果缺少 P 操作，则不能保证对临界资源的互斥访问；如果缺少 V 操作，会使临界资源得不到释放，从而使因等待该资源释放而阻塞的进程不能被唤醒。

注意：通常情况下，用于表示临界资源互斥访问的信号量初值一般为 1，仅允许一个进程进入临界区。

3.1.4 利用信号量解决进程同步问题

利用信号量可以实现进程或语句之间的前趋关系，即实现进程之间的直接制约关系。设有两个并发执行的进程 P_1 和 P_2，P_1 中有语句 S_1，P_2 中有语句 S_2。如果希望 S_1 执行完成后才能执行 S_2，则只需为 P_1、P_2 设置一个公有信号量 T，并设其初值为 0，将 $V(T)$ 放在 S_1 之后，将 $P(T)$ 放在 S_2 之前，即可实现 $S_1 \rightarrow S_2$ 的前趋关系。

进程 P_1：S_1；$V(T)$。

进程 P_2：$P(T)$；S_2。

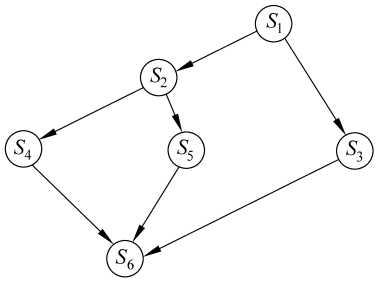

图 3.4 前趋关系图举例

若先执行 P_2 进程，由于 T 初值为 0，则 $P(T)$ 使得进程 P_2 阻塞，只有 P_1 执行完 $V(T)$ 后，P_2 才能被唤醒，否则一直处于阻塞状态；若先执行 P_1 进程，则当 P_1 执行完 $V(T)$ 后，$T=1$，P_2 中的 $P(T)$ 不会造成 P_2 阻塞，从而实现了 $S_1 \rightarrow S_2$ 的前趋关系。

前趋关系图中每个前趋关系都用一个信号量实现，该信号量由前趋关系中的两个结点共享，信号量初始值一般为 0。

例题 3.2 某前趋关系如图 3.4 所示。

针对图中的前趋关系 $S_1 \rightarrow S_2$、$S_1 \rightarrow S_3$、$S_2 \rightarrow S_4$、$S_2 \rightarrow S_5$、$S_3 \rightarrow S_6$、$S_4 \rightarrow S_6$、$S_5 \rightarrow S_6$，分别设置信号量 a、b、c、d、e、f、g，初始值均为 0。实现该前趋关系图的伪代码如下：

```
semaphore var a,b,c,d,e,f,g=0,0,0,0,0,0,0;
begin
    parbegin            //并发程序开始
    begin S1; V(a); V(b); end;
    begin P(a); S2; V(c); V(d); end;
    begin P(b); S3; V(e);  end;
    begin P(c); S4; V(f);  end;
    begin P(d); S5; V(g);  end;
    begin P(e); P(f); P(g); S6;  end;
    parend              //并发程序结束
end
```

前趋关系是进程之间的一种同步关系。在实现同步关系时，应注意同一个信号量的 P、V 操作也要成对出现，不过 P 操作和 V 操作出现在不同进程的代码中。

信号量虽然能够实现并发进程之间的同步和互斥，但必须合理地使用该工具才能既保证进程的同步和互斥，又保证各程序正确执行完毕。例如上例中，如果用户在

```
begin S1; V(a); V(b); end;
```

中忘记书写 $V(b)$，则会将导致进程 S_3 和 S_6 无法执行，永久处于等待状态。

除了实现进程的前趋关系外，信号量还能实现进程之间控制消息的传递。

例题 3.3 现实生活中，在公共汽车上，司机进程和售票员进程各司其职，两者的工作流程如图 3.5 所示。

司机在正常行车中，售票员售票，两者之间没有制约关系，可以任意并发。但在其他环节，司机进程和售票员进程之间存在着如下同步关系。

(1) 司机停车后，必须在等待售票员关闭车门后才能启动车辆。售票员在关车门后给司机发信号，允许司机开车。

(2) 售票员售完票后阻塞，等待司机给他发送到站信号，只有收到到站信号后才能开车门。

采用 P、V 原语表示，司机进程和售票员进程之间的同步关系如图 3.6 所示。

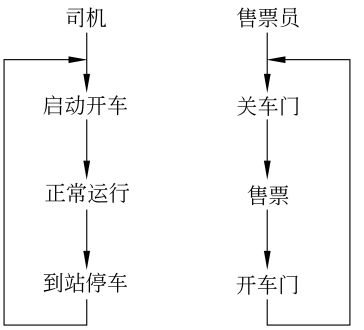

图 3.5　司机和售票员的工作流程

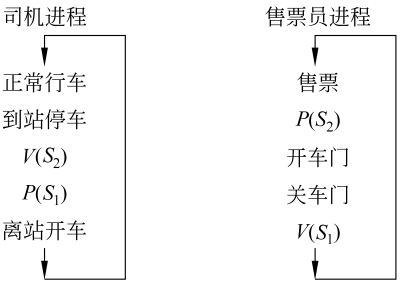

图 3.6　P、V 原语表示的司机进程和售票员进程之间的同步关系

在图 3.6 中，设置信号量 S_1 表示是否允许司机启动汽车，其初始值为 0；设置信号量 S_2 表示是否允许售票员开车门，其初始值为 0。

注意：通常情况下，表示进程之间同步关系的信号量初值一般为 0。当信号量值大于 0 时，表示可用资源的数目；当信号量值为 0 时，表示没有可用资源，但还没有进程阻塞等待；当信号量小于 0 时，其绝对值表示在该信号量阻塞队列中阻塞进程的个数。

司机进程和售票员进程的伪代码如下：

```
var S1,S2:semaphore:=0,0;
begin
    parbegin
    司机进程:
    begin
        while (true)
        {
            P(S1);          //等待售票员发送关门信息
            启动车辆;
            正常行车;
            到站停车;
            V(S2);          //给售票员发送到站信息
        }
    end;
    售票员进程:
    begin
        while (true)
        {
```

```
            关车门;
            V(S1);          //给司机发送关门信息
            售票;
            P(S2);          //等待司机发送到站信息
            开车门;
            上下乘客;
        }
        end;
    parend;
end.
```

试依据所学知识,分析伪代码是否满足司机进程和售票员进程之间的同步关系。

例题 3.4 P_1、P_2、P_3 3 个进程合作完成一项任务。它们都需要通过同一台设备输入各自的数据 a、b、c。该输入设备必须互斥地使用,而且其第 1 个数据必须由 P_1 进程读取,第 2 个数据必须由 P_2 进程读取,第 3 个数据必须由 P_3 进程读取。3 个进程分别对输入数据进行下列计算。

$$P_1: x = a + b;$$
$$P_2: y = a * b;$$
$$P_3: z = y + c - a$$

最后,P_1 进程通过所连接的打印机将计算结果 x、y、z 的值打印出来。试用信号量实现它们间的同步。

本题中,为了控制 3 个进程依次使用输入设备进行输入,必须分别设置 3 个信号量 S_1、S_2、S_3。其中,S_1 的初始值为 1,S_2 和 S_3 的初始值为 0。使用上述信号量后,3 个进程不会同时使用输入设备,故不必再为输入设备配置互斥信号量。另外,还需要设置信号量 S_b、S_y、S_z 表示数据 b 是否已经输入,以及 y、z 是否已经计算完成。若它们的初值均为 0,3 个进程的动作可描述如下:

```
P1()
{
    P(S1);
    从输入设备输入数据 a;
    V(S2);
    P(Sb);
    x=a+b;
    P(Sz);
    使用打印机打印出 x、y、z 的结果;
}
P2()
{
    P(S2);
    从输入设备输入数据 b;
    V(S3);
    V(Sb);
    y=a*b;
    V(Sy);
```

```
}
P3()
{
    P(S3);
    从输入设备输入数据 c;
    V(S1);
    P(Sy);
    z=y+c-a;
    V(Sz);
}
```

3.2 典型进程同步问题详解

本节介绍 3 个经典的进程同步问题，通过对这 3 个问题的分析和求解，进一步介绍用 P、V 原语实现进程间的同步和互斥机制。

3.2.1 生产者-消费者问题

生产者-消费者问题是最典型的进程同步问题之一，由荷兰计算机科学家 E. W. Dijkstra 首先提出。实际上，计算机系统中的许多问题都可看作生产者-消费者问题或其扩展问题。生产者-消费者问题描述了一组生产者进程向一组消费者进程提供产品，两类进程共享一个由 n 个缓冲区组成的有界缓冲池，生产者进程向空缓冲区中放入产品，消费者进程从放有数据的缓冲区中取出产品并消费掉。假定生产者进程和消费者进程互相等效，只要缓冲池未满，生产者进程就可以把产品放入缓冲池；只要缓冲池未空，消费者进程便可以从缓冲池中取走产品。禁止生产者进程向满的缓冲池再输送产品，也禁止消费者进程从空的缓冲池中提取产品。

生产者-消费者问题是常见的相互合作进程的一种抽象，有很大的代表性和实用价值。在实际操作系统中存在大量的类似实例。例如，在输入数据时，输入进程是生产者进程，计算进程是消费者进程；在打印时，计算进程是生产者进程，而打印进程是消费者进程。

缓冲池具有 n 个缓冲区，常被组织成一个数组。每个缓冲区都能存入一个产品，缓冲池中的所有缓冲区构成一个循环缓冲结构，如图 3.7 所示。输入指针 in 指向下一个可存放产品的空缓冲区，输出指针 out 指向下一个可从中获取产品的满缓冲区。在循环缓冲结构中，输入指针加 1 表示为 in＝(in＋1)％n，输出指针加 1 表示为 out＝(out＋1)％n。当 in＝out时，表示缓冲池空，没有可供消费者进程消费的产品；当(in＋1)％n＝out 时，表示缓冲池满，没有可供生产者进程存放数据的空缓冲区。生产者进程使用局部变量 product_good 暂存每次刚生产出的产品，消费者进程使用局部变量 consume_good 暂存每次取出来消费的产品。

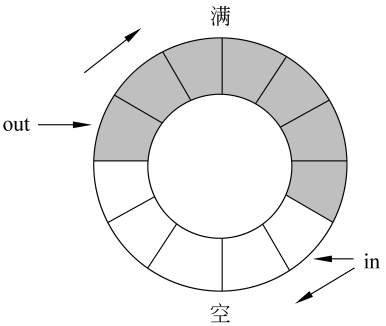

图 3.7　环形缓冲池

经过分析可知，生产者-消费者问题中，存在两个同步要求，分别介绍如下。

(1) 缓冲池空时不能消费，必须先生产后消费。解决办法是设置一个信号量 full，其初始值为 0。消费者进程中在消费操作前先执行 P(full)操作，如果没有产品，则在该 P 操作上阻塞，消费者进程进入 full 的阻塞队列中。生产者进程在生产完产品后，执行 V(full)操作，通知消费者进程可以消费产品，或把 full 的阻塞队列中的消费者进程唤醒。

(2) 缓冲池满时不能再往其中放产品。生产和消费要平衡，不能只生产不消费。为了防止出现“只生产不消费”现象，解决办法是设置一个新的信号量 empty，其初值为缓冲区的个数 n。生产者进程在生产前先执行 P(empty)操作，如果没有空闲缓冲区，则阻塞并进入 empty 所指的阻塞队列中，不再继续生产，防止了只生产不消费现象的出现；消费者进程在消费产品后，执行 V(empty)操作，通知生产者进程可以继续生产产品或把 empty 的阻塞队列中的生产者进程唤醒。

当消费掉所有的产品后，信号量 full 的值会变成 0，消费者进程在执行 P(full)操作时会阻塞，不会出现只消费不生产现象。因此，无须再设置新信号量防止出现只消费不生产现象。

此外，生产者-消费者问题中还存在一个互斥要求，即对缓冲池操作时，各个进程之间要互斥。解决办法是设置一个信号量 mutex，其初始值为 1。在生产者进程和消费者进程中，在缓冲区的操作前用 P(mutex)、操作后有 V(mutex)，如图 3.8 所示。

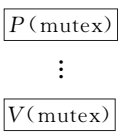

图 3.8　生产者进程和消费者进程中缓冲区的操作前后

综上所述，在生产者-消费者问题中需要设置 3 个信号量：信号量 empty 和 full 用于实现同步要求，信号量 mutex 用于实现互斥要求。该问题的伪代码如下：

```
var mutex,empty,full:semaphore :=1,n,0;   //信号量 mutex 实现缓冲区操作互斥要求
//信号量 empty 表示空缓冲区的个数,信号量 full 表示满缓冲区的个数,用于实现同步要求
    buffer:array[0,…,n-1] ofitem;          //用数组表示缓冲池,数组长度为缓冲区的个数
    in,out: integer :=0,0;                 //in 表示空缓冲区指针,out 表示满缓冲区指针
                                  //in 和 out 这两个变量只是普通的整型变量,不是信号量
    parbegin
Producer()                                 //生产者进程
{
    while (1)
    {
        …
        produce a product_good;
        …
        P(empty);                          //获得一个空缓冲区
        P(mutex);                          //互斥使用缓冲区
        buffer[in]=product_good;
```

```
        in=(in+1)%n;                          //把产品 product 放入缓冲区中
        V(mutex);                             //释放缓冲区
        V(full);                              //缓冲池得到一个满缓冲区
        until false;
    }
}
Consumer()                                    //消费者进程
{
    while (1)
    {
        P(full);                              //获得一个满缓冲区
        P(mutex);                             //互斥使用缓冲区
        consume_good=buffer[out];
        out=(out+1) %n;                       //把产品从缓冲区中取走
        V(mutex);                             //释放缓冲区
        V(empty);                             //缓冲池得到一个空缓冲区
        consume the consume_good;
        until false;
    }
}
    parend
```

在每个进程中,用于实现互斥的 P(mutex)和 V(mutex)必须成对出现,且出现在同一进程中。用于实现同步的信号量 empty 和 full 的 P、V 操作也需成对出现,但同一个信号量的 P、V 操作位于不同进程中。

注意:无论是在生产者进程还是在消费者进程中,V 操作的次序无关紧要,但每个进程中多个 P 操作的先后次序不能随意颠倒,否则可能出现死锁现象。

一般情况下,先执行对同步信号量的 P 操作,然后再执行互斥信号量的 P 操作。例如,在生产者-消费者问题中,Producer()进程中的 P(empty)和 P(mutex)互换先后次序。先执行 P(mutex),假设成功,生产者进程获得对缓冲区的访问权,但如果此时缓冲池已满,没有空缓冲区可供其使用,会导致后续的 P(empty)原语没有通过,Producer()在信号量 empty 上阻塞,而此时 mutex 已被改为 0,已不能恢复成初值 1。

操作系统把 CPU 使用权切换到消费者进程后,Consumer()进程执行 P(full)成功,但其执行 P(mutex)时由于进程 Producer()正在访问缓冲区,所以不成功,阻塞在信号量 mutex 上。生产者进程和消费者进程均阻塞,无法继续执行,相互等待对方释放资源,故产生死锁现象。

注意:P(empty)和 P(mutex)互换次序,本质上是扩大了信号量 mutex 控制的临界区,把 P(empty)也纳入了信号量 mutex 控制的临界区,这导致生产者进程和消费者进程之间存在死锁的风险。

下面分析一个较为复杂的生产者-消费者问题,加深对该类问题的理解。

例题 3.5 某寺庙,有小和尚、老和尚若干。有一个水缸,由小和尚提水入缸供老和尚饮用。水缸可容 10 桶水,水取自同一口井中。井口径窄,每次只能容下一个桶取水。水桶总

数为 3 个。每人一次从缸中取水仅为 1 桶，且不可同时进行。试用记录型信号量给出有关取水、入水的算法描述。

本题是生产者-消费者问题的扩展。小和尚为生产者，从井中提水倒入缸中；老和尚为消费者，从缸中取水饮用。本题中水缸相当于生产者-消费者模型中的缓冲池。

根据题意，定义信号量及其初始值如下。

(1) 水桶为临界资源需互斥使用，定义信号量 bucket，因有 3 个桶，故初始值为 3。

(2) 水井一次只能允许一个桶取水，定义互斥信号量 well，初始值为 1。

(3) 水缸一次只能允许一个人取水，定义互斥信号量 jar，初始值为 1。

(4) empty 和 full 用于小和尚和老和尚之间的同步制约关系。因为水缸能存 10 桶水，所以 empty 初始值为 10；开始时缸中没有水，full 的初始值为 0。

算法描述如下：

```
semaphore bucket=3,jar=1,full=0,empty=10,well=1;
void little_monk()              //小和尚入水算法
{
    while(1)
    {
        P(empty);
        P(bucket);
        P(well);
        从水井中打水;
        V(well);
        P(jar);
        倒入水缸;
        V(jar);
        V(full);
        V(bucket);
    }
}

void old_monk()                 //老和尚取水算法
{
    while (1)
    {
        P(full);
        P(bucket);
        P(jar);
        从缸中取水;
        V(jar);
        V(empty);
        从桶中倒入饮用;
        V(bucket);
    }
}
```

思考：如果颠倒小和尚进程 little_monk()中的 P(empty)和 P(bucket)和老和尚进程

old_monk()中的 P(full)和 P(bucket),会存在什么风险?

3.2.2 哲学家就餐问题

哲学家进餐问题是在 1965 年由 Dijkstra 提出并解决的一个典型进程同步问题。该问题是进程并发控制问题中的一个典型例子,也极具代表性。

哲学家进餐问题描述:有 5 个哲学家倾注毕生精力用于思考和吃饭,他们围坐在一张圆桌旁,在圆桌上有 5 个盘子和 5 把叉子,如图 3.9 所示。每位哲学家的行为通常是思考,当其感到饥饿时,便试图取其左右最靠近他的叉子进餐。由于食物的原因,哲学家只有同时拿到两把叉子后才能进餐,进餐完后,释放两把叉子并继续思考。

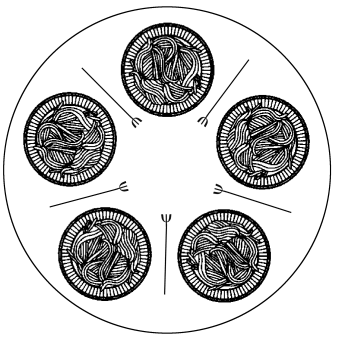

图 3.9 哲学家就餐问题

相邻两个哲学家对位于两人之间的叉子存在竞争关系。叉子为临界资源,必须互斥使用;每位哲学家必须获得两把叉子后才能进餐。为了实现对叉子的互斥使用,需要为 5 把叉子分别设置一个互斥信号量,初值均为 1。为了简化信号量的定义,可把这 5 个信号量定义成信号量数组。

哲学家就餐问题描述如下:

```
semaphore fork[5]={1,1,1,1,1};
第 i 位哲学家的活动可描述为(i=0,1,2,3,4):
void philosopher(int i)
{
    while (true)
    {
        think();
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat();
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
```

当上述进程并发运行时,假如 5 位哲学家同时饥饿而各自拿起左边的叉子时,就会使信号量数组中的 5 个信号量都变为 0,当他们试图去拿右边的叉子时,都因为无叉子可拿而陷入无限期等待,等待其他哲学家释放叉子,出现死锁现象。

为了解决上述问题,避免哲学家进程间出现死锁,可采取以下方案进行解决。

(1) 至多只允许 4 个哲学家同时进餐,以保证至少有一个哲学家能够进餐,最终总会释放出他所使用过的两把叉子,从而可使更多的哲学家进餐。

增加一个信号量 room,初始值为 4,表示只允许 4 个哲学家同时进入餐厅就餐,这样就能保证至少有一个哲学家可以获得两把叉子就餐。而申请不到叉子的哲学家进入信号量 room 的阻塞等待队列。依据先进先出原则,先申请的哲学家会较先吃饭,吃完饭后释放叉子,这样就不会出现某个哲学家饿死的现象。具体如下:

```
semaphore fork[5]={1,1,1,1,1};
semaphore room=4;                    //信号量 room 的初始值为 4,比哲学家总数少 1
第 i 位哲学家的活动可描述为(i=0,1,2,3,4):
void philosopher(int i)
{
    while (true)
    {
        think();
        P(room);                     //请求进入房间进餐
        P(fork[i]);                  //请求左手边的叉子
        P(fork[(i+1)%5]);            //请求右手边的叉子
        eat();
        V(fork[(i+1)%5]);            //释放右手边的叉子
        V(fork[i]);                  //释放左手边的叉子
        V(room);                     //退出房间释放信号量 room
    }
}
```

(2) 规定奇数号哲学家先拿他左边的叉子,然后再去拿右边的叉子;而偶数号哲学家则相反。按此规定,奇数号(1,3)的哲学家先拿左边的叉子,偶数号(0,2,4)的哲学家先拿右边的叉子,最后总会有一位哲学家能获得两把叉子进餐。申请不到叉子的哲学家进入阻塞等待队列,当其他哲学家进程执行完毕、释放叉子时,会被唤醒,因此不会出现饿死的哲学家进程。具体如下:

```
semaphore fork[5]={1,1,1,1,1};
第 i 位哲学家的活动可描述为(i=0,1,2,3,4):
void philosopher(int i)
{
    while (true)
    {
        think();
        if (i/2==0)                  //偶数哲学家,先右后左
        {
            P(fork[(i+1)%5]) ;
            P(fork[ i]) ;
        }
        else                         //奇数哲学家,先左后右
        {
            P(fork[i]);
            P(fork[(i+1)%5)]);
        }
        eat();
        V(fork[i]);
        V(fork[(i+1)%5)]);
    }
}
```