# 第3章 进程同步

处理器(CPU)是计算机系统最重要的资源,为了使CPU和各种系统资源得到充分利用,操作系统中引入了多道程序设计的概念。在多道程序设计系统中,多个程序同时装入内存,使得程序或程序段能够并发执行,从而共享计算机系统的软硬件资源,提高了系统资源的利用率。但这些程序的并发执行对程序之间的执行顺序、资源使用等带来了诸多的影响,使得程序的设计和系统管理变得更为复杂,由此也在操作系统中引入了进程的概念。本章将对并发进程之间的制约关系以及进程同步问题进行阐述。

## 3.1 同步与互斥的基本概念

在多道程序设计环境中,多个进程的并发执行提升了系统资源利用率,但也带来了进程间相互作用的复杂性。进程不再是孤立运行的实体,它们可能因共享资源或协作完成任务而产生相互制约关系。这种制约关系若不加以有效管理,会导致进程执行结果不确定、资源竞争冲突甚至系统死锁等问题。因此,深入理解进程间的同步与互斥机制,是保障多道程序系统高效、正确运行的核心前提。本节将从程序的执行方式入手,分析顺序执行与并发执行的本质区别,进而阐述进程同步与互斥的基本概念、临界资源与临界区的管理原则,以及同步机制需遵循的核心准则,为后续同步机制的实现与经典问题的解决奠定理论基础。

### 3.1.1 程序顺序执行

**1. 程序与顺序性**

程序是实现算法功能的一组指令序列,人们在计算机上通过编写程序来实现相应的功能。在单道程序设计环境中,程序都是顺序执行的,即在内存仅装入一道程序,让其独占系统中的所有资源,一个程序执行完毕后,另一个程序才能执行。

为了更直观清晰地描述程序间执行的顺序关系,我们首先介绍一个图示工具——前驱图(Precedence Graph)。前驱图是一个有向无环图(Directed

Acyclic Graph,DAG),用于描述程序段或进程之间执行的先后顺序关系。前驱图由节点和有向边组成：节点可以表示一条语句、一个程序段或一个进程;有向边表示两个节点之间存在的偏序(Partial Order)或前驱关系(Precedence Relation)。

前驱关系可以用"→"来表示。例如,节点 $P_i$、$P_j$ 之间存在前驱关系,我们可以表示为($P_i$、$P_j$)∈→,或者 $P_i \rightarrow P_j$,表示 $P_i$ 必须在 $P_j$ 执行之前完成。这时我们称 $P_i$ 是 $P_j$ 的直接前驱,而 $P_j$ 是 $P_i$ 的直接后继。在前驱图中,没有前驱的节点称为初始节点(Initial Node),没有后继的节点称为终止节点(Final Node)。在图 3-1 所示的前驱图中存在以下的前驱关系：$P_1 \rightarrow P_2$,$P_1 \rightarrow P_3$,$P_1 \rightarrow P_4$,$P_2 \rightarrow P_5$,$P_3 \rightarrow P_6$,$P_4 \rightarrow P_6$,$P_5 \rightarrow P_7$,$P_6 \rightarrow P_7$。另外,前驱图严格遵循无环性原则以防止逻辑矛盾,所以在前驱图中不允许出现环。

例如,在单道程序系统中有多个程序,程序在执行时,通常先输入用户程序和数据,然后CPU 对输入的数据进行处理,最后输出处理结果。即每个程序都由三个程序段组成,在运行时按照"输入($I$)→计算($P$)→输出($O$)"的顺序一步步执行各个程序段。前一个程序段执行结束后,才能开始下一个程序段的执行,即这三个程序段之间存在着前驱关系。其执行顺序如图 3-2 所示。

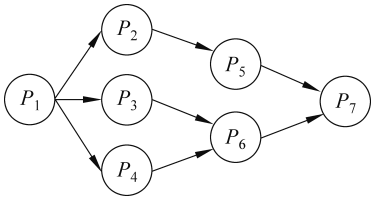


图 3-1　前驱图

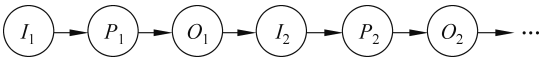


图 3-2　程序顺序执行

顺序程序设计是把一个程序设计成一个顺序执行的指令序列,使程序按照预定的逻辑顺序依次执行。这里提到的顺序的含义不仅指一个程序模块的内部,也指两个程序之间。一个程序在处理器上的执行是严格有序的,只有当一个操作结束后,才能开始后继操作,这被称为程序执行的内部顺序性。如果一个进程包含若干程序,这些程序按照调用的次序严格有序执行,则被称为程序执行的外部顺序性。

**2. 程序顺序执行时的特性**

将程序设计成顺序执行,一个程序的不同程序段在执行时都要按照次序的顺序先后执行,当前一个程序段执行完才能执行下一个程序段。不同程序也按照先后顺序执行。程序顺序执行与其速度无关,即程序的最终输出仅与初始输入数据有关,而与时间无关。程序顺序执行具有以下几个特征。

(1) 执行的顺序性：一个程序在处理器上是严格按照顺序执行的,每个操作必须在下一个操作开始之前结束。

(2) 环境的封闭性：程序运行时独占受操作系统保护的资源,资源状态只能由程序本身决定和改变,不受外界因素影响。

(3) 结果的可再现性：程序对相同数据集的执行轨迹是确定的,程序执行的结果与执行速度和执行时段无关。

程序顺序执行方便了程序员开发程序,程序调试也变得简单方便,但这种方式执行效率低,并且浪费系统资源,为此,现代操作系统普遍支持多道程序设计。在多道程序处理系统

中，内存中可以同时装入多个程序，多个程序可以共享资源并发执行。但是程序的并发执行与顺序执行有显著的区别，使程序的设计和系统管理变得复杂，因此在操作系统中引入了进程的概念。

### 3.1.2　程序并发执行

为了提高系统的处理能力和资源的利用率，现代计算机系统中普遍引入了多道程序设计技术。在多道程序系统中，多个程序是并发执行的，即一个程序执行未结束，另一个程序就开始执行，这样程序外部顺序性的特性就消失了。

**1. 程序的并发执行**

程序并发执行指一组程序的执行在时间上是重叠的，即在同一时间间隔内运行多个程序。对于单 CPU 系统来说，内存中可以存在多个程序，而 CPU 与外部设备、外部设备与外部设备之间可以并行操作。从宏观上看，系统中有多道程序在同时运行；从微观上看，这些程序在 CPU 上是交替执行的。如果是多处理器系统，这些程序是可以并行执行的。我们这里所说的程序并发执行一般指多个程序在单 CPU 系统交替执行。程序并发执行时，各个程序会以各自独立的、不可预知的速度向前推进。这样可能造成一些程序在执行后得到不可预知或错误的结果。

下面还是以一个常见的例子来说明程序的并发执行的情况。如图 3-3 所示的多道程序并发执行，第一个程序的输入程序段 $I_1$ 完成数据输入后，处理程序段 $P_1$ 对输入数据进行处理，与此同时第二个程序的输入程序段 $I_2$ 也在输入数据，这样第一个程序的处理程序段 $P_1$ 与第二个程序的输入程序段 $I_2$ 在时间上就是重叠的。接下来，第三个程序的输入程序段 $I_3$ 开始输入数据，同时第二个程序的处理程序段 $P_2$ 对输入数据进行处理，而第一个程序的输出程序段 $O_1$ 开始输出处理结果，即 $I_3$、$P_2$ 和 $O_1$ 可以并发执行，并以此类推，多个程序的不同程序段会并发执行。

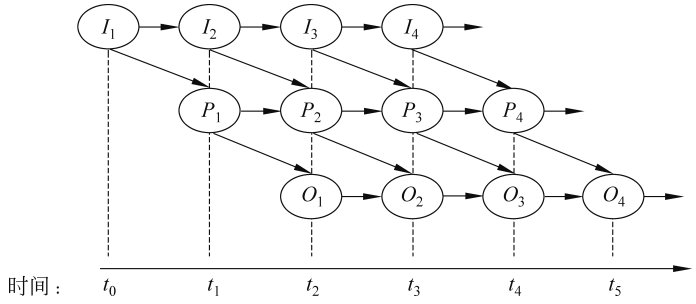


**图 3-3　多道程序并发执行**

同时，我们也可以看出，第二个程序中 $P_2$ 的执行必须满足：$I_2$ 执行完成，并且第一个程序的 $P_1$ 也执行完成并释放 CPU。此时，这几个程序的执行已不再是一个封闭的环境，程序的并发执行使得制约的条件增加了。因此，程序并发执行时系统资源（硬件和软件资源）就不再被某个程序独占，而是由多个并发执行的程序所共享。这样一方面提高了系统资源的利用率，但另一方面也造成了多个并发程序对资源的竞争，进而导致程序执行环境和运行速度的改变，可能会导致程序运行结果不确定的问题。

**2. 程序并发执行时的特征**

(1) 间断性。

程序在并发执行时,因为要共享资源,但是资源往往都少于正在执行的程序的需求数,所以会存在资源抢占的问题。因此,每个程序在 CPU 上运行时,都是时断时续的。当一个资源被占用时,其他需要该资源的程序不得不暂停,等资源被释放时才可以执行。

(2) 失去封闭性。

程序正在并发执行时,由于它们共享资源或者合作完成同一项任务,系统的状态不再受其中一个程序的控制和改变,所以就失去了封闭性。

(3) 不可再现性。

因为程序在并发执行时失去了封闭性,所以任何一个程序都有可能对系统的状态进行改变,这就意味着即使初始条件和运行环境不变,程序多次运行的结果也可能会各不相同。

程序在顺序执行时具有顺序性、封闭性和可再现性的特征,使程序和程序的执行过程之间具有一一对应的关系,因此用“程序”这个静态概念完全可以代替“程序执行过程”这个动态概念。但在多道程序环境下,程序并发执行失去了封闭性,具有了间断性,程序运行的结果也具有了不可再现性。这时就无法再使用程序的运行对程序进行描述,由此在操作系统中引入了“进程”的概念,使用动态的“进程”对参与并发执行的每个程序(含数据)进行准确的描述。

### 3.1.3 进程同步

**1. 两种进程的制约关系**

在一个多道程序系统中运行的并发进程通常会有多个,这些并发进程可能是无关的,也可能是交互的。如果一组并发进程分别在不同的变量集合上操作,任何一个进程的执行都不依赖其他进程,并且与其他进程的进展情况无关,即它们是各自独立的,则这些并发进程相互之间是无关的。显然,无关的并发进程,也是相互独立的进程,一定没有共享的变量,它们分别在各自的数据集合上操作。而如果一个进程的执行依赖其他进程的进展情况,或者说,一个进程的执行可能影响其他进程的执行结果,则说明这些并发进程是交互的。

在多道程序环境中,交互的并发进程由于共享 CPU、I/O 设备等系统资源,或者为完成某个共同任务而相互合作,进程之间会存在以下两种不同形式的相互制约关系。

(1) 进程同步。

进程同步又称直接相互制约关系,是指为完成某个共同的任务而建立的两个或多个进程,这些进程因为需要在某些位置上协调它们的工作次序而产生的制约关系。进程间的直接制约关系就源于它们之间的相互合作。

(2) 进程互斥。

进程互斥又称间接相互制约关系,是指进程之间由于共享某些独占型资源所产生的相互制约关系。各个并发执行的进程不可避免地需要共享一些系统资源(如打印机、磁带机等设备),为了保证多个进程对该类资源的互斥访问,形成间接制约的关系。

进程同步和互斥并不是相互对立的,进程互斥是多个进程争夺独占型资源产生的竞争制约关系;进程同步表示多个进程可以同时执行,只不过有速度上的差异,需要速度上匹配,不存在资源是否独占或共享的问题。我们可以将进程互斥看作进程同步的一种特例。

**2. 与时间有关的错误**

对于交互的并发进程来说，可能有若干并发进程同时使用共享资源，即一个进程一次使用尚未结束，另一个进程已经开始使用，形成交替使用共享资源的现象。而有的进程由于自身或外界的原因而可能被中断，且断点是不固定的。至于一个进程被中断后，哪个进程可以先运行，而被中断的进程在什么时候再去占用处理器等问题，则与进程调度策略有关。这些进程执行的速度不能由自身控制，而进程的执行结果又依赖进程的时序，如果不加以控制，在共享资源(变量)时就会出现错误而得不到唯一的结果，或者处于永远等待资源的状态。这样的一些错误我们统称为“与时间有关的错误”。

下面通过一个简单的例子来说明。例如，有一个飞机票售票系统有两个终端，并发运行两个售票程序 $T_1$ 和 $T_2$，并共享一个公共变量 $n$，$n$ 中存放航班余票数。程序描述如下：

| 终端程序 $T_1$： | 终端程序 $T_2$： |
|---|---|
| …<br>x=n;<br>if (x>=1){<br>  x--;<br>  n=x;<br>  售出一张机票;}<br>else<br>  机票已售完; | …<br>y=n;<br>if (y>=1){<br>  y--;<br>  n=y;<br>  售出一张机票;}<br>else<br>  机票已售完; |

程序 $T_1$ 和 $T_2$ 的执行都以各自独立的速度向前推进，它们的语句在时间上可以出现交叉执行。假设，在售票前，终端程序 $T_1$ 和 $T_2$ 同时查询到公共变量 $n$ 的值，并分别向旅客售出一张机票，而实际余票数只减去 1。这样一张机票就卖给了两位旅客，显然，这样的结果是错误的。

产生这种情形的根本原因在于：在并发程序中共享了公共变量，使得程序的计算结果与并发程序执行的速度有关。这种错误的结果又往往是与时间有关的，随执行速度的不同而异，所以把它称为“与时间有关的错误”。

### 3.1.4　临界区与临界资源

**1. 临界资源**

在多道程序系统中，并发执行的进程不可避免地需要共享一些系统资源(如内存、打印机、摄像头等)。实际上，诸如各种物理设备、变量、数据、内存缓冲区等资源，在一个时间段内只允许一个进程使用，这类资源称为临界资源(critical resource)。一方面，并发执行的进程需要共享资源；另一方面，临界资源的访问又必须是互斥地进行(不能同时共享)，很显然，这会导致资源访问上的矛盾。所以，我们要通过互斥的方式来解决此类问题，防止两个或两个以上的进程同时访问临界资源。

对临界资源的访问，需要互斥地进行，即同一时间段内只能允许一个进程访问该资源。当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源。

**2. 临界区**

由上述可知，无论是硬件临界资源还是软件临界资源，多个进程必须互斥地对它们进行访问。在每个进程中，我们将访问临界资源的那段代码称为临界区(critical section)。若不

同进程的临界区使用的是同一个临界资源，则称它们的临界区为相关临界区（或同类临界区）。显然，如果能保证各个进程互斥地进入相关临界区，就可以实现进程对临界资源的互斥访问（后面对于临界区的访问控制均指的是相关临界区）。这样，实现进程对临界资源的互斥访问就转变为进程互斥地进入临界区。要求每个进程在进入临界区之前，需要先对临界资源进行检查。如果此刻临界资源未被访问就允许对临界区的访问，并设置它正被访问的标志；如果此刻该临界资源正在被某进程访问，则本进程不能进入临界区；进程访问完临界资源后需释放资源，并清除被访问的标志。

为此，需要在临界区前面增加一段用于检查临界区是否被访问的进入区（entry section）代码；在临界区的后面需添加一段恢复临界区访问标志的退出区（exit section）代码；进程其余部分的代码则称为剩余区（remainder section）。临界区是进程中访问临界资源的代码段，而进入区和退出区是负责实现互斥的代码段。访问临界资源的进程的代码结构描述如下：

```
while(TRUE)
{
    进入区
    临界区
    退出区
    剩余区
}
```

**3. 同步机制遵循的规则**

为了实现进程互斥地进入临界区，可在系统中设置专门的同步机构来协调各个进程间的运行。所有同步机制都应遵循以下 4 条规则。

（1）空闲让进：当临界区空闲时，应允许一个请求进入临界区的进程立即进入临界区。

（2）忙则等待：当已有进程进入临界区时，其他试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。

（3）有限等待：对任何请求访问临界资源的进程，应保证其能在有限时间内能进入临界区，以免进入“死等”状态。

（4）让权等待：当进程不能进入临界区时，应立即释放占有的处理器，使其他进程有机会得到处理器的使用权，防止进程进入“忙等”状态。

## 3.2 基本实现方法

进程的同步机制包括软件同步机制和硬件同步机制。软件同步机制是指通过软件编程手段实现进程同步的方法，即通过软件层面的逻辑控制来协调进程之间的同步关系。硬件同步机制则是利用计算机硬件提供的指令来实现进程同步。通过软件同步机制和硬件同步机制，可以实现对临界区的互斥访问，并保证程序并发执行时的可再现性。

### 3.2.1 *软件方法*

在操作系统的进程同步研究中，软件同步机制主要是通过在进入区和退出区设置和检查标志来实现互斥的，防止多个进程同时进入临界区。临界区问题的软件解决方案具有重要的理论价值。下面介绍两个经典的互斥算法——Dekker 算法与 Peterson 算法，它们通过

巧妙的标志位设计实现了进程间的互斥访问，为理解现代同步机制奠定了基础。

**1. Dekker 算法**

Dekker 算法由荷兰数学家 Theodorus Jozef Dekker 于 1969 年提出，用于解决双并发进程的同步与互斥问题。Dekker 算法结合了两种基本策略：轮转法（Turn-based）和标志位法（Flag-based），有效解决了进程间的互斥与饥饿问题。

Dekker 算法通过设置两个标志变量 flag[0]和 flag[1]，分别记录两个进程的状态，即是否准备进入临界区。此外，设置一个轮转变量 turn 来决定哪个进程有进入临界区的优先权。进程在访问临界区之前会先检查另一个进程是否也需要使用临界区，如果另一个进程也需要使用临界区，并且拥有访问权限，这时当前进程就会等待另一个进程访问临界区完毕；若另一个进程无法同时满足这两个条件，那么当前进程就会直接访问临界区。Dekker 算法的基本思路如下。

（1）进程设置自己的请求标志 flag 为 1，表明请求进入临界区。

（2）若进程检测到另一个进程的标志 flag 为 1，则检查轮转权 turn 归属，并主动让出竞争等待轮转权变更。

（3）若进程检测到另一个进程的标志 flag 为 0，则该进程进入临界区执行。

（4）进程执行完毕退出临界区，重置自己的 flag 为 0，并切换 turn 的值。

Dekker 算法实现的伪代码描述如下：

```
int flag[2]={0,0};      //标志数组: flag[i]表示进程 i 是否请求进入临界区
int turn=0;             //轮转变量: 指示当前允许进入临界区的进程 ID(0 或 1)
```

```
void P0( ){
  flag[0]=1;            //设置自身请求标志
  while(flag[1]!=0){    //若 P1 请求
    flag[0]=0;          //主动让步
    if(turn!=0) while(1);
                        //轮转权在 P1,等待
    flag[0]=1;          //重新设置请求标志
  }
  访问临界区;
  turn=1;               //移交轮转权给 P1
  flag[0]=0;            //清除自身请求标志
}
```

```
void P1(){
  flag[1]=1;            //设置自身请求标志
  while(flag[0]!=0){    //若 P0 请求
    flag[1]=0;          //主动让步
    if(turn!=1) while(1);
                        //轮转权在 P0,等待
    flag[1]=1;          // 重新设置请求标志
  }
  访问临界区;
  turn=0;               //移交轮转权给 P0
  flag[1]=0;            //清除自身请求标志
}
```

虽然 Dekker 算法首次完整解决了双进程互斥问题，满足了互斥性、有限等待和空闲让进的原则，但无法避免忙等现象，降低了处理器的效率，且实现的复杂度较高（包含多层循环和标志重置）。

**2. Peterson 算法**

Peterson 算法是一个经典的解决两个进程间互斥问题的算法，由 Gary L.Peterson 于 1981 年提出。Peterson 算法是一个实现互斥锁的并发程序设计算法，在保持正确性的前提下简化了实现逻辑，可以控制两个进程访问一个共享的互斥资源而不发生访问冲突。相比于 Dekker 算法，Peterson 算法解决了互斥访问问题，而不需要像 Dekker 算法一样强制轮流访问，可以按正常顺序进行工作。

Peterson 算法使用两个标志变量 flag[0]和 flag[1]来表示进程是否准备进入临界区，同时使用一个共享变量 turn 来指示哪个进程优先进入临界区。其算法基本思路如下。

(1) 每个进程在请求进入临界区时,设置自己的标志 flag 为 true,表示其希望进入临界区。

(2) 一个进程设置变量 turn 为另一个进程的编号,表示自己愿意让另一个进程先进入临界区。

(3) 一个进程在进入临界区之前,需要检查另一个进程的标志 flag 和变量 turn,确保不会与另一个进程冲突。

(4) 当一个进程退出临界区时,将自己的标志 flag 设置为 false,表示其已经不再需要进入临界区。

Peterson 算法实现的伪代码描述如下:

```
boolean flag[2]={ false, false};    //标志数组: flag[i]为 true 表示进程 i 请求进入临界区
int turn;                           //轮转变量: 指示当前允许进入临界区的进程 ID(0 或 1)
```

```
void P0()
{
      while(true)
      {
      flag[0] =true; //设置自身请求标志
      turn =1;        //让 P1 先进入临界区
      while(flag[1] && turn ==1);
                //P1 请求且允许 P1 进入,等待
      临界区;
      flag [0]=false;   //清除自身请求标志
    }
}
```

```
void P1()
{
    while(true)
    {
    flag[1]=true;   //设置自身请求标志
    turn=0;         //让 P0 先进入临界区
    while(flag[0] && turn ==0);
                //P0 请求且允许 P0 进入,等待
    临界区;
    flag[1]=false; //清除自身请求标志
    }
}
```

Dekker 算法和 Peterson 算法是互斥算法设计的里程碑: Dekker 算法首次验证了纯软件方案的可行性,Peterson 算法展示了通过精妙的条件设计简化实现,共同揭示了原子操作的重要性,为硬件同步指令(如 Test-and-Set)的研发提供了启示。在现代操作系统中,这些算法已逐渐被硬件支持的同步原语取代,但其设计思想仍然重要,帮助理解互斥问题的本质特征,展示了软件方法实现互斥的可行性,推动了后续锁机制的研究。

通过分析这两个经典算法,揭示了进程同步的核心挑战与解决方案。这些早期研究成果不仅具有历史价值,其蕴含的设计思想对理解现代操作系统的同步机制仍具有重要指导意义。

### 3.2.2 硬件方法

在软件同步机制里,可以将标志看作锁,如果在测试和关锁之间发生中断,会导致无法互斥进入临界区。若需在此期间禁止进程中断,可以通过计算机提供的一些硬件指令实现对临界区的管理。

**1. 关中断**

在硬件实现的互斥机制中,最简单直接的就是在进程进入临界区时"关中断",退出临界区时"开中断"。关中断后,进程在临界区执行期间,计算机系统不响应中断,从而不会引发进程调度,也就不会发生进程或线程切换。关中断机制的结构描述如下:

```
while(TRUE)
{
    关中断;
```

```
    临界区;
    开中断;
    其余部分;
}
```

使用开/关中断的方法是非常简单有效的,但其也存在一些弊端。

(1) 临界区代码必须要简短,否则,关中断时间过长会影响系统效率。

(2) 关中断权限一旦被滥用会给系统带来危险。

(3) 关中断方法只适用于单处理器,不适用于多处理器系统。一个处理器关中断后,其他处理器上的进程仍然可以访问临界资源。

**2. TS 指令**

进程在测试与关锁之间有可能发生中断,那我们在测试之前关闭中断(操作系统内核不响应中断信号),到测试并关锁后再打开中断。这样可以保证两个操作之间的连续性,保证临界资源的互斥访问。通常可以使用一条硬件指令——"测试与设置"指令 TS(Test-and-Set)来实现临界资源的互斥访问。在 TS 指令中设置一个整型变量 $x$,将 $x$ 看作锁。若 $x$ 值为 0 表示开锁状态,此时没有进程访问临界区;$x$ 值为 1 表示关锁状态,此时临界区被某个进程占用。TS 指令描述如下:

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

通过 TS 指令可以实现临界区的开锁和关锁的原语操作。进程在进入临界区之前先用 TS 指令测试 $x$,如果其值为 0,则表示没有进程使用临界资源,可以进入临界区,并将 1 赋值给 $x$;如果 $x$ 值为 1,则表示临界资源被其他进程使用,不能进入临界区。由于 TS 指令执行过程不能被中断,因此本方法能够保证实现进程互斥。

利用 TS 指令实现进程同步的代码描述如下:

```
do{
    进入区;
    while TS(x);                    //加锁
    临界区;
    x=0;                            //解锁
    剩余区;
}while(true);
```

**3. Swap 指令**

Swap 指令又称对换指令,用于交换两个字的内容。Swap 指令的代码描述如下:

```
void Swap(boolean * a, boolean * b)
    boolean temp;
    temp = * a;
    * a = * b;
    * b =temp;
}
```

可以为每个临界资源设置一个初值为 false 的全局布尔变量 $x$,在每个进程中使用一个

局部布尔变量 $y$。Swap 指令实现进程互斥的代码描述如下：

```
do{
    y =true;
    do{
        Swap(&x,&y);
    }while(y!=false);              //加锁
    临界区;
    x =false;                      //解锁
    剩余区;
}while(true);
```

上述硬件指令实现简单有效，但是当临界资源忙碌时，其他访问进程必须不断地调用硬件进行测试，处于一种“忙等”状态，不符合“让权等待”的原则，容易造成处理器时间的浪费。

## 3.3 信号量机制

前面介绍的解决进程互斥的方法中，软件方法比较复杂，效率较低；硬件指令的方法实现简单，但当有进程在临界区内执行时，其他进程需要不断测试，消耗较多的处理器时间。另外，上述方法很难解决复杂的进程同步问题。

### 3.3.1 信号量

1965 年，荷兰学者 E.W.Dijkstra 提出利用信号量(Semaphore)机制解决进程同步问题。信号量被用于在进程间进行信息传递。在长期的使用中，信号量机制得到了很大的发展，从整型信号量到记录型信号量，再发展到信号量集。从取值上来看信号量可以分为二值信号量和一般信号量。现在信号量机制已被广泛地用于单处理器和多处理器系统以及计算机网络中。

在操作系统中，信号量是表示物理资源的实体，通常表示为一个与队列有关的整型变量。与一般整型变量不同，信号量只能进行三种操作：除初始化外，还有 wait 和 signal 两个标准的原语操作。wait()操作和 signal()操作通常又分别被称为 P、V 操作(名称来自荷兰语 Proberen(检测)和 Verhogen(增量))，另外也可以用 up 和 down 来表示。

**1. 整型信号量**

Dijkstra 最初把信号量定义为一个用于表示资源数目的整型信号量，代表资源的数目或可同时使用该资源的进程个数。

信号量 $S$ 的 wait()操作和 signal()操作的功能描述如下：

```
void wait(S){
    while(S<=0);
    S--;
    }
```

```
void signal(S){
    S++;
    }
```

在 wait(S)操作中，如果信号量 $S>0$，则 $S$ 减 1；如果信号量 $S\leqslant 0$，则会不断地测试 $S$，直至 $S>0$。因此该机制并未遵循“让权等待”原则，而是使进程处于“忙等”状态。

**2. 记录型信号量**

为解决整型信号量的“忙等”问题，需遵循“让权等待”原则。然而，这会导致多个并发进程请求访问临界资源时，仅有一个进程能获得信号量，其余进程均需阻塞等待。为此，在整

型信号量基础上增加一个等待进程链表指针 list，用于链接所有等待进程，从而形成记录型信号量的数据结构。因此这种信号量也被称为记录型信号量，其描述如下：

```
typedef struct{
    int value;
    struct PCB *list;
} semaphore;
```

其中，value 是信号量值，值为正时表示可用资源的数量；值为负时表示资源全部分配完毕，等待资源的进程在等待队列中排队。如果 value 的初值为 1，则表示该资源为临界资源，此信号量称为互斥信号量。

记录型信号量 S 的 wait()操作和 signal()操作定义如下：

```
void wait(semaphore S){
    S.value--;
    if(S.value<0){
        add this process to S.list;
        block(S.list);
    }
}
```

```
void signal(semaphore S){
    S.value++;
    if(S.value<=0){
        remove a process P from S.list;
        wakeup(P);
    }
}
```

wait(S)操作用于进入临界区前进行资源申请。每执行一次 wait(S)操作意味着请求一个单位的该类资源，S.value 的值减 1。若 S.value＜0，则说明该类资源已分配完毕，此时进程将调用 block 原语，进行阻塞自己，放弃处理器，并在该类资源的等待队列中排队。该机制遵循了“让权等待”原则，此时 S.value 的绝对值表示在等待该类资源的进程数。

signal(S)操作用于释放资源。每执行一次 signal(S)操作表示进程释放一个单位该类资源，使系统中该类资源的可供分配个数加 1，即 S.value＋＋。若 S.value≤0，则说明还有进程在等待队列中等待该类资源，此时还需要调用 wakeup 原语来将 S.list 中的一个等待进程唤醒。

### 3.3.2　信号量的应用

**1. 利用信号量实现互斥**

利用信号量机制可以很容易解决进程互斥问题。要实现多个进程互斥访问临界区，需要为该临界区设置一个互斥信号量 mutex，信号量初值设为 1，然后将各个进程的临界区置于 wait(mutex)和 signal(mutex)之间。信号量实现进程互斥的代码结构为：

```
wait(mutex);
  临界区;
signal(mutex);
```

例如，有两个进程 $P_1$ 和 $P_2$，欲访问同一个临界资源，则设置一个初值为 1 的互斥信号量为 S，然后将临界区放在 wait(S)和 signal(S)之间，即可实现两个进程互斥访问临界区。相关代码描述如下：

```
semaphore S=1; //互斥信号量
cobegin
    Process P1(){
        wait(S);
        临界区;
        signal(S);
      }
    Process P2(){
        wait(S);
        临界区;
        signal(S);
      }
coend
```

假设进程 $P_1$ 先执行，由于信号量 $S$ 的初值为 1，执行 wait(S)操作后 $S$ 的值为 0，说明 $P_1$ 成功申请到临界资源，可以进入临界区执行。如果此时进程 $P_2$ 请求进入临界区，执行 wait(S)操作后，$S$ 的值由 0 变为－1，进程 $P_2$ 被阻塞。当进程 $P_1$ 执行完毕退出临界区，执行 signal(S)操作后，临界资源被释放，$S$ 的值由－1 变为 0，将唤醒被阻塞的进程 $P_2$，$P_2$ 可以进入临界区。同理，如果进程 $P_2$ 先执行也能够实现进程的互斥。

需要注意的是，通过信号量机制实现进程互斥时，互斥信号量的初值应设为 1，一个信号量的 wait()和 signal()操作必须在同一个进程中成对出现。缺少 wait()操作就无法实现临界资源的互斥访问；缺少 signal()操作则会导致资源永不被释放，等待进程将陷入永远等待。

**2. 利用信号量实现同步**

信号量机制是一种有效的进程同步手段，能够确保多个进程按照既定顺序执行。以两个并发进程 $P_1$ 和 $P_2$ 为例，其中 $P_2$ 的代码段 $C_2$ 必须在 $P_1$ 的代码段 $C_1$ 之后运行。为此，首先需要明确两个代码段的执行顺序，即 $P_1$ 的代码段 $C_1$ 先执行(称为前操作)，$P_2$ 的代码段 $C_2$ 后执行(称为后操作)。接下来，定义一个初始值为 0 的信号量。在前操作 $C_1$ 执行完毕后调用 signal 操作；而在后操作 $C_2$ 执行之前调用 wait()操作，则可以实现进程的同步关系。相关代码描述如下：

```
semaphore S=0;
cobegin
    Process P1(){                    Process P2(){
        C1;                              wait(S);
        signal(S);                       C2;
    }                                }
coend
```

设置同步信号量 $S$ 的初值为 0，取值范围为(－1，0，1)。假设 $P_2$ 先执行，执行到 wait(S)处时，由于此时 $S$ 值为 0，根据 wait()操作的定义可知，$P_2$ 会被阻塞；而 $P_1$ 执行完代码段 $C_1$ 后，会执行 signal(S)，唤醒处在阻塞队列中的 $P_2$，于是 $P_2$ 将从阻塞队列移入就绪队列中，被调度执行，则可以执行代码段 $C_2$。使用 wait()和 signal()操作实现进程同步时，信号量的初值由用户根据资源数量进行设置。

当需要双向同步(例如，进程 $P_1$ 和进程 $P_2$ 需交替执行)时，需分别设置一个信号量以控制不同方向的同步。在进程同步问题中，对同一个信号量的 wait()和 signal()操作分布在两个不同的进程中，才能实现双向同步控制。

**3. 利用信号量实现前驱关系**

信号量可以描述复杂的前驱关系，多个合作进程完成一个任务，这些合作进程的执行有的可以并发，有的存在先后次序。为了保证该顺序可以正确执行，每一对前驱关系应该设置一个初始值为 0 的信号量，并在“前操作”之后调用信号量的 signal()操作，在“后操作”之前调用信号量的 wait()操作。例如，图 3-4(a)所示的前驱图中共有七对前驱关系，分别为 $S_1 \rightarrow S_2$、$S_1 \rightarrow S_3$、$S_2 \rightarrow S_4$、$S_2 \rightarrow S_5$、$S_3 \rightarrow S_6$、$S_4 \rightarrow S_6$ 和 $S_5 \rightarrow S_6$。为实现所有前驱关系，需要设计 7 个初值为 0 的信号量，如图 3-4(b)所

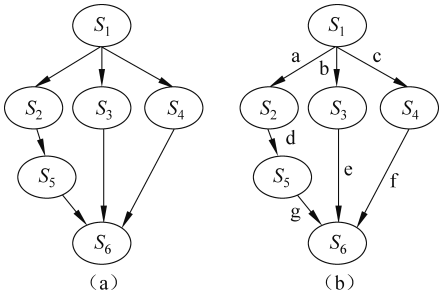


**图 3-4　前驱图示例**

示，并针对每一对前驱关系分别设置 wait()操作和 signal()操作。图 3-4 表示的前驱关系实现算法描述如下：

```
semaphore a =b =c =d =e =f =g =0;
cobegin
    P1(){S1;signal(a);signal(b); signal(c)}
    P2(){wait(a);S2; signal(d);}
    P3(){wait(b);S3;signal(e);}
    P4(){wait(c);S4;signal(f);}
    P5(){wait(d);S5;signal(g);}
    P6(){wait(e);wait(f); wait(g);S6;}
coend;
```

## 3.4　经典同步问题

在多道程序环境下，进程同步问题十分重要，由此也产生了一系列经典的同步问题。接下来将通过对这些经典同步问题的解决更好地理解进程同步。

### 3.4.1　生产者-消费者问题

生产者-消费者问题是操作系统中进程协作关系的经典抽象模型，同时包含同步与互斥机制。该模型的核心是：生产者进程生成数据项并放入共享缓冲区，消费者进程则从缓冲区取出并使用这些数据项。通过这种方式，生产者-消费者问题解决了一类并发进程在共享资源访问中的协调问题。

问题描述：一组生产者进程和一组消费者进程共享初始为空、大小为 $n$ 的缓冲区，只有缓冲区不空时，生产者才能把产品放入缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。缓冲区是临界资源，它只允许一个生产者放入产品，或者一个消费者从中取出产品。

下面对生产者-消费者问题进行分析。

(1) 生产者进程和消费者进程之间存在同步关系。当缓冲区有空，生产者才能将生产的产品放入缓冲区中；当缓冲区不为空(有产品)，消费者才能从缓冲区中取产品。

(2) 生产者进程和消费者进程之间存在互斥关系。共享缓冲区是一个临界资源，生产者和消费者进程对共享缓冲区的使用必须互斥进行。

在生产者-消费者问题中，生产者、消费者以及缓冲区都可以是一个或多个，因此可能出现的情况有多种。下面我们就以其中的两种情况为例进行讨论，其余情况可以参照方法予以解决。

**1. 单生产者单消费者共享单个缓冲区问题**

根据生产者进程和消费者进程之间的同步关系，设置两个同步信号量：Sempty 表示空缓冲区的数量，即缓冲区可放产品数量；Sfull 表示满缓冲区的数量，即缓冲区可用产品数量。在只有一个缓冲区的情况下，缓冲区初始是空的，因此设置 Sempty 初值为 1，Sfull 初值为 0。生产者进程在向缓冲区中放入一个产品前，需要执行 wait(Sempty)申请一个空闲缓冲区，放入产品后，执行 signal(Sfull)释放一个装有产品的缓冲区；消费者进程在取产品前，需要执行 wait(Sfull)申请一个装有产品的缓冲区，取出产品后，执行 signal(Sempty)释放一个空闲缓冲区。实现单生产者单消费者共享单个缓冲区的实现代码如下：

```
int Buffer;
semaphore Sempty =1;        //同步信号量,空闲缓冲区的数量
semaphore Sfull =0;         //同步信号量,装有产品的缓冲区数量,即产品的数量
main(){
    cobegin
      producer();
      consumer();
    coend
}
```

```
Process producer(){
  while(true){
      produce an item;
      wait(Sempty);   //申请一个空闲缓冲区
      Buffer =item;
      signal(Sfull);
                  //释放一个装有产品的缓冲区
    }
}
```

```
Process consumer(){
    while(true){
      wait(Sfull);   //申请装有产品的缓冲区
      item =Buffer;
      signal(Sempty);   //释放一个空闲缓冲区
      consumer an item;
    }
}
```

因为存在两个同步关系,可以看到在生产者、消费者进程中,同一信号量的 wait()和 signal()操作是成对出现的,并且分布在两个不同进程中。因为单缓冲区内仅允许单一操作(生产者或消费者独占),所以不需要额外的互斥信号量。

**2. 多生产者多消费者共享多个缓冲区问题**

下面考虑有 $m$ 个生产者、$n$ 个消费者和 $K$ 个共享缓冲区的问题,这类问题的核心在于协调三类冲突。

(1) 生产者-生产者冲突:多个生产者同时写入导致数据覆盖。

(2) 消费者-消费者冲突:多个消费者同时读取可能导致数据重复读取。

(3) 生产者-消费者冲突:缓冲区满时生产者阻塞,缓冲区空时消费者阻塞。

为此,将缓冲区抽象为 $K$ 个单元的环形队列 Buffer,定义生产者写入位置指针 in 和消费者读取位置指针 out(in 和 out 初值相同)。

(1) 每个生产者进程将产品放入 Buffer[in],然后,采用模运算:in = (in + 1) % $K$ 移动指针。

(2) 每个消费者进程从 Buffer[out]取走产品,然后,采用模运算:out = (out + 1) % $K$ 移动指针。

(3) 为了避免多个生产者同时写入导致数据覆盖,将生产者对 Buffer[in]的操作和移动指针 in 定义为临界区。

(4) 为了避免消费者同时读取导致数据重复,将消费对 Buffer[out]的操作和移动指针 out 定义为临界区。

生产者-消费者的同步控制与单生产者单消费者共享单个缓冲区问题一致,只是初始状态下,空缓冲区的数量为 $K$。缓冲区是临界资源,必须互斥使用,因此要设置一个互斥信号量 mutex,初值为 1。多生产者多消费者共享多个缓冲区问题的实现代码如下:

```
int Buffer[K];
semaphore mutex =1;         //互斥信号量
semaphore Sempty =K;        //同步信号量,空闲缓冲区的数量
semaphore Sfull =0;         //同步信号量,产品的数量,非空缓冲区的数量
int in =0, out =0;
```

```
main(){
    cobegin
      producer_i();        //i=1～m
      consumer_j();        //j=1～n
    coend
}
```

```
Process producer_i (){   //i=1～m
    while(true){
      produce an item;
      wait(Sempty);   //申请一个空闲缓冲区
      wait(mutex);
      Buffer[in] =item;//把产品放入缓冲区
      in = (in +1) %K;     //移动指针
      signal(mutex);
      signal(Sfull);      //释放一个产品
    }
}
```

```
Process consumer_j (){  //j=1～n
    while(true){
      wait(Sfull);       //申请一个产品
      wait(mutex);
      item =Buffer[out];   //从缓冲区取出产品
      out = (out +1) %K;  //移动指针
      signal(mutex);
      signal(Sempty);    //释放一个空闲缓冲区
      consumer an item;
    }
}
```

在生产者进程和消费者进程中可以看到，都是先执行对同步信号量(Sempty、Sfull)的 wait()操作，再执行对互斥信号量 mutex 的 wait()操作。这里的两个 wait()操作次序不能互换。假设在生产者进程中先执行 wait(mutex)，生产者进程获得对缓冲区的访问权，如果此时缓冲区不空，在执行 wait(Sempty)时生产者进程将在信号量 Sempty 上阻塞，而消费者进程由于互斥使用的原因，不能获得缓冲区的访问权，两个进程都无法执行，相互等待对方释放资源，出现永远等待的现象。

### 3.4.2　哲学家进餐问题

哲学家进餐问题(Dining Philosophers Problem)是由 E. W. Dijkstra 提出的经典的同步问题之一。如图 3-5 所示，有 5 位哲学家围坐在一张圆桌旁，相邻两人间仅放置 1 根筷子，全桌共 5 根筷子。哲学家持续在思考状态与饥饿状态间转换：饥饿时必须同时持有左、右两侧的筷子才能进入用餐状态；用餐结束后立即释放筷子并进入思考状态。

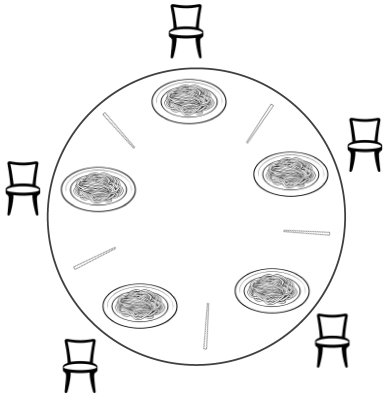

图 3-5　哲学家进餐问题

由于每位哲学家只能取自己两侧的筷子，因此，桌上的 5 根筷子显然不属于同类资源。根据信号量的定义，需要定义 5 个初值为 1 的信号量来代表 5 根筷子。哲学家在进餐使用两个 wait()操作分别申请左侧和右侧的筷子，用餐完毕，使用两个 signal()操作释放筷子。由此，哲学家进餐问题可以使用如下伪代码来描述：

```
semaphore chopstick[5]={1, 1, 1, 1, 1}        //对应 5 根筷子
Process philosopher ()_i{                     //i=0～4
    while (true) {
        思考;
        wait(chopstick[i]);                   //取左侧筷子
        wait(chopstick[(i +1)%5]);            //取右侧筷子
        用餐;
        signal(chopstick[i]);
        signal(chopstick[(i +1)%5]);
```

```
        }
}
```

若5位哲学家同时饥饿而各自拿起了左侧的筷子，当他们试图去拿起右侧的筷子时，都将因无筷子而无限期地等待下去。哲学家进餐问题反映了在进程同步中处理不当可能引起永远等待的现象。解决这个问题的方法如下。

**1. 不允许多位哲学家同时取筷子**

在哲学家就餐问题中，每位哲学家必须依次获取左、右两根筷子才能进餐。问题在于，每位哲学家获取第一根筷子后，操作系统可能调度切换到另一位哲学家执行。如果这种情况持续发生，最终可能导致所有哲学家都只持有了一根筷子，并等待获取第二根筷子而陷入永远等待的状态。

这是因为每个哲学家进程在执行 wait(chopstick[i])(获取左侧筷子)后，没有立即连续执行 wait(chopstick[(i + 1)%5])(获取右侧筷子)，而是被中断，让其他哲学家进程有机会执行它们自己的 wait(chopstick[i])操作。这种并发执行的调度方式使得所有哲学家进程都处在只获得一根筷子而等待另一根筷子的状态。

为了解决这个永远等待的问题，可以将哲学家获取两根筷子的整个操作过程(即从获取第一根筷子开始到获取第二根筷子结束)定义为一个临界区，并确保各个哲学家进程互斥地执行这个临界区内的代码。按照前面进程互斥的实现方法，只需定义一个初值为1的信号量 mutex，并在每个进程进入临界区前执行 wait(mutex)，退出临界区后执行 signal(mutex)，则可以解决哲学家永远等待的问题。具体实现代码如下：

```
semaphore chopstick[5]={1, 1, 1, 1, 1};        //每根筷子设置为互斥量
semaphore mutex=1;                             //对应5根筷子
Process philosopher ()_i{                      //i=0～4
    while (true) {
        思考;
        wait(mutex);
        wait(chopstick[i]);                    //取左侧筷子
        wait(chopstick[(i +1)%5]);             //取右侧筷子
        signal(mutex);
        用餐;
        signal(chopstick[i]);
        signal(chopstick[(i +1)%5]);
        }
}
```

**2. 最多允许4位哲学家同时用餐**

为了解决哲学家永远等待的问题，可以限制同时尝试获取筷子的哲学家数量。假设最多只允许4位哲学家同时尝试进餐，那么即使这4位哲学家都成功拿起了左侧的筷子，由于总共只有5根筷子，必定至少有一位哲学家能够成功获得右侧筷子并完成进餐。

实现方法：设置一个初值为4的信号量 count，用于控制允许尝试进餐的哲学家数量。每位哲学家在尝试获取筷子前，必须先获得一个“进餐许可”；用餐结束后再释放该许可。具体实现代码如下：

```
semaphore chopstick[5]={1, 1, 1, 1, 1};        //对应5根筷子
semaphore count=4;                             //最多允许4位哲学家同时尝试进餐
Process philosopher ()_i{                      //i=0～4
    while(true){
        思考;
```

```
        wait(count);
        wait(chopstick[i]);                    //取左侧筷子
        wait(chopstick[(i +1)%5]);             //取右侧筷子
        用餐;
        signal(chopstick[i]);
        signal(chopstick[(i +1)%5]);
        signal(count);
    }
}
```

**3. 限制哲学家取筷子的顺序**

为了解决哲学家永远等待的问题，可以为 5 位哲学家先按 0～4 的顺序进行编号，再为奇、偶编号的哲学家设定不同的取筷顺序。

(1) 奇数编号哲学家(1 号、3 号)：先取左侧筷子，再取右侧筷子。

(2) 偶数编号哲学家(0 号、2 号、4 号)：先取右侧筷子，再取左侧筷子。

这种差异化策略打破了对称性竞争关系：相邻哲学家可能会首先竞争同一根筷子而导致一个哲学家阻塞；所有哲学家不会同时请求同一方向的筷子；当所有哲学家同时取第一根筷子，由于取筷方向不同会出现的情况如下。

(1) 0 号和 1 号哲学家竞争 1 号筷子，必定导致一个哲学家阻塞。

(2) 2 号和 3 号哲学家竞争 3 号筷子，必定导致一个哲学家阻塞。

(3) 4 号哲学家独占 0 号筷子(无竞争)。

这种策略确保至少一位哲学家能获得两根筷子，打破永远等待的僵局。具体实现代码如下：

```
semaphore chopstick[5]={1, 1, 1, 1, 1}         //每根筷子设置为互斥量
Process philosopher ()_i{                      //i=0～4
    while(true){
        思考;
        if(i %2==1){                           //奇数号哲学家
            wait(chopstick[i]);                //取左侧筷子
            wait(chopstick[(i +1)%5]);         //取右侧筷子
        }
        else if(i %2 ==0){                     //偶数号哲学家
            wait(chopstick[(i +1) %5]);        //取右侧筷子
            wait(chopstick[i]);                //取左侧筷子
        }
        用餐;
        signal(chopstick[i]);                  //放下筷子不必考虑先后
        signal(chopstick[(i +1)%5]);
    }
}
```

### 3.4.3　读者-写者问题

读者-写者问题描述的是：有两组并发进程，共享一个文件，我们把只读该文件的进程称为“读者”，其他进程为“写者”。多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者在读文件时写者也不能写文件。读者-写者问题其实是保证一个写者进程必须与其他进程互斥地访问共享对象的同步问题。读者-写者问题的具体要求如下。

(1) 允许多个读者可以同时对文件执行读操作。

(2) 只允许一个写者往文件中写信息。

(3) 任一写者在完成写操作之前不允许其他读者或写者工作。

在读者-写者问题中,读者进程与写者进程之间存在三种制约关系。

(1) 读者与读者之间允许同时读。

(2) 读者与写者之间需要互斥。

(3) 写者与写者之间需要互斥。

为了解决读者进程、写者进程之间的同步关系。在实际应用中有读者优先、写者优先等策略。

**1. 读者优先**

读者优先规定,当存在读者在读文件时,新来的读者可直接读文件;而写者必须等待所有读者(包括后续到达的读者)完成读文件后才能写文件。

根据分析,需要设置如下几个信号量。

(1) readcount:计数器,用于记录当前正在读的读者进程个数,readcount 赋初值为 0,仅在 readcount=0 时才允许写者进程访问文件。

(2) rmutex:读互斥信号量,用于读者进程互斥访问公用变量 readcount,初值为 1。

(3) wmutex:写互斥信号量,用于读者进程与写者进程的互斥,以及写者进程与写者进程的互斥,初值为 1。

实现读者优先的读者-写者问题的伪代码如下:

```
semaphore rmutex =1, wmutex =1;
int readcount =0;          //读者进程计数
```

```
Process Reader()_i{     //i=1～m
    while(true) {
        wait(rmutex);   //阻塞其他读者进程
        if(readcount ==0)  //第一个读进程
            wait(wmutex);   //阻塞写进程
        readcount ++;
        signal(rmutex);
        读文件;
        wait(rmutex);
        readcount --;
        if(readcount ==0)
            signal(wmutex);
        signal(rmutex);
    }
}
```

```
Process Writer()_j{     //j=1～n
    while(true) {
        wait(wmutex);   //阻塞其他进程
        写文件;
        signal(wmutex);
    }
}

main()
{
    cobegin
        Reader()_i;     //i=1～m;
        Writer()_j;     //j=1～n;
    coend
}
```

在读者优先算法中,只要有一个读者在进行读文件操作,那么后续来的读者都将被允许访问文件,从而导致写者长时间等待,出现“饥饿”现象。因此读者优先算法适用于读者进程较多,而写者进程较少的场景。这种算法对于写者进程较多、读者进程较少的场景是不利的。

**2. 写者优先**

写者优先规定,当有写者在写文件时,读者进程必须等待所有写者(包括后续到达的写者)完成写操作后才能读文件。

写者优先算法在读者优先算法的基础上,增加一个同步信号量 readsem,用于阻塞所有的读者,初值为 1。增加一个计数器,即整型变量 writecount,记录正在等待的写者数目,只

有当 writecount＝0 时，才可以释放等待的读者。writecount 为多个写者共享的变量，是临界资源，用互斥信号量 wc_mutex 控制，wc_mutex 初值是 1。当第一个写者进程到来时，通过 wait(readsem)阻止后续到达的读者进程。

实现写者优先的读者-写者问题的伪代码如下：

```
semaphore rmutex =1, wmutex =1, wc_mutex =1, readsem =1;
int readcount =0, writecount =0;
```

```
Process Reader()_ i{      //i=1～m
    while(true) {
        wait(readsem); //若有写者,阻塞读者
        wait(rmutex);
        if(readcount ==0)
            wait(wmutex); //阻止新写者进入
        readcount ++;
        signal(rmutex);
        signal(readsem);
        读文件操作;
        wait(rmutex);
        readcount --;
        if(readcount ==0)
            signal(wmutex);
        signal(rmutex);
                        //运行其他读者访问
    }
}
main()
{
    cobegin
        Reader()_i;       //i=1～m;
        Writer()_j;       //j=1～n;
    coend
}
```

```
Process Writer()_ j{       //j=1～n
    while(true) {
        wait(wc_mutex);      //写者互斥进入
        writecount++;
        if(writecount ==1)
            wait(readsem);
                            //阻塞新读者进入
        signal(wc_mutex);
        wait(wmutex);
        执行写操作;
        signal(wmutex);
        wait(wc_mutex);
        writecount --;
        if(writecount ==0)
            signal(readsem);
                            //唤醒被阻塞读者
        signal(wc_mutex);
      }
  }
```

写者优先算法中，读者首先通过 wait(readsem)检查是否有写者。若有写者(readsem 被占用)，读者会被阻塞；若没有写者，读者才能继续。写者优先算法适合写者进程较多，而读者进程较少的场景。

### 3.4.4　睡眠理发师问题

理发店里有一位理发师、一张理发椅和 $N$ 张供等候理发的顾客坐的椅子。如果没有顾客，理发师就在理发椅上睡觉；如有顾客需要理发，他必须唤醒理发师；如果理发师在忙且有空闲椅子，那么顾客会坐下来等待；如果所有椅子都不空闲，那么顾客离去。

要解决睡眠理发师问题，我们设置两类进程，一个理发师进程，一组顾客进程。再分析一下问题中的同步互斥关系。

(1) 理发师和顾客之间是同步关系，理发师等待顾客来，然后为顾客理发，若理发师在睡觉，顾客需要唤醒他，执行上有先后顺序，需设置两个同步信号量。

(2) 顾客对椅子的操作又是互斥的，属于竞争关系，所以需要互斥信号量来保证椅子的数量准确。

为了实现对人数的增加和减少，设置一个互斥信号量来对人数进行互斥访问。实现睡眠理发师问题的伪代码如下：

```
semaphore customers =0;
semaphore barbers =0;
semaphore mutex =1;
int waiting =0;                          //等待理发的顾客数
int chairs =N;                           //为顾客准备的椅子数
cobegin
    process barber(){                    //理发师
        while(true){
            wait(customers);             //等待顾客,有顾客则继续执行,否则睡眠
            wait(mutex);
            waiting--;
            signal(barbers);             //通知等待在该条件的顾客可以理发了
            signal(mutex);
            cut_hair();                  //理发
        }
    }
    process customer_i(){                //顾客
        wait(mutex);
        if(waiting <chairs){             //判断是否有空椅子
            waiting++;
            signal(customers);           //唤醒理发师
            signal(mutex);
            wait(barbers);               //理发师忙,顾客坐下等待
            get_haircut();               //否则,顾客得到理发
        }
        else{
            signal(mutex);               //无空椅子,顾客离开
            leave();
        }
    }
coend
```

## 3.5 管　　程

信号量机制有效地解决了并发进程之间的同步与互斥问题。但对临界区的执行分散在各个进程中,不利于系统对临界资源的管理,同时也很难发现和纠正分散于各个用户进程中同步原语的错误使用。于是提出了管程的概念。

### 3.5.1 管程的概念

20 世纪 70 年代初,P. B. Hansen 和 C. A. R. Hoare 提出了另一种同步机制——管程(Monitor)。管程的基本思想是将分散在各进程中的控制和管理临界资源的临界区集中起来统一管理。建立一个管程来统一管理各个进程的临界区,既解决了信号量机制中同步操作分散、管理复杂并可能引发系统死锁的问题,又保证了进程的互斥访问。

**1. 管程的定义**

管程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,它提供一种互斥机制,进程可以互斥地调用这些过程。Hansen 对管程的定义为“一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作,这组操作能同步进程和改变管程中的数据”。

由管程的定义可知,管程由四部分组成:管程的名称;局部于管程的共享数据结构说明;对该数据结构进行操作的一组过程;对局部于管程的共享数据初始化语句。通俗地说,