第 3 章 并发控制——互斥与同步

进程和线程是现代操作系统的重要概念，操作系统设计的核心是进程管理，主要涉及下述三方面：

(1) 多道程序，即单个处理机中多个进程的管理。大部分个人计算机、工作站、单处理器系统上的操作系统，例如 Windows 98、OS/2 和 Macintosh System 7 都支持多道程序，共享单处理机系统，UNIX 也支持多道程序。

(2) 多进程，即多个处理机中多个进程的管理。多处理机一般用于较大的系统，诸如 MVS 和 VMS 之类的操作系统支持多进程。服务器和工作站也开始支持多进程，Windows NT 就是一个为这种环境而设计的操作系统。

(3) 分布式进程，即运行于分布式计算机系统中的多个进程的管理。

上述三方面和操作系统设计的共同基础是并发，并发包含了大量的设计问题，包括进程间通信、竞争和共享资源，多个活跃进程的同步和处理机时间的分配等。这些问题并非仅存在于多进程和分布式进程系统中，也存在于单处理机多道程序的系统中。

在以下三种情况下可能出现并发：

(1) 多个应用。多道程序用于多个作业或多个应用程序动态地共享处理机时间。

(2) 结构化应用。如果多个应用程序遵循结构化程序设计和模块化设计的原则，那么它们就可以作为并发进程高效地实现。

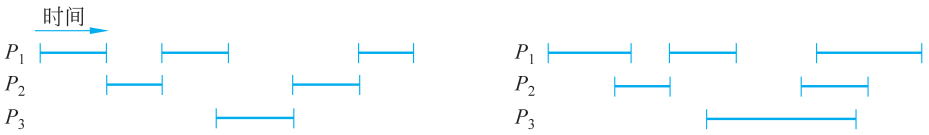
(3) 操作系统结构。如果将结构化的优点应用到操作系统中，那么这些操作系统本身也可作为一个并发进程集高效地实现。

本章首先介绍并发的概念和并发进程的实现。支持并发的基本要求是互斥，即若一个进程正在执行，则其他所有进程将被排斥执行；本章的第二部分讨论实现互斥的可能途径，所有方法都是通过软件加以解决的，并且要用到“忙-等”(Busy Waiting)技术。由于这些方法过于复杂，而且“忙-等”方法也不理想，这就要寻找不包含“忙-等”的方法，这些方法还需要得到操作系统的支持或用编译器实现。本章讨论了三种不包含“忙-等”的方法：信号量(semaphores)、管程(monitors)和消息传递(message passing)。

本章还用两个经典问题描述了这些概念并比较了这些方法的优劣，它们是生产者/消费者(producer/consumer)问题和读者/写者(readers/writers)问题。

3.1 并发原理

在一个单处理机多道程序的系统中，进程被分隔插入处理机时间中交替执行，使进程表面看起来是同时执行的，如图 3.1(a)所示，尽管这不是真正的并行，并且进程间的调度将带来额外的开销，但插入执行的主要好处是提高了执行效率，在多处理机系统中不仅可以插入执行，还可以重叠执行，如图 3.1(b)所示。



(a) 插入执行　　(b) 插入和重叠执行

图 3.1　并发进程的执行

插入执行和重叠执行看起来好像是两种根本不同的执行方式，事实上它们都是并发执行方式。以单处理机系统为例，多道程序中进程执行的相对速度无法预测的问题主要和以下几方面相关：其他活动进程的状况，操作系统处理中断的方式，操作系统的调度规则。此外，还存在如下困难：

(1) 全局变量的共享充满危险。如果两个进程使用同一全局变量，并对此变量进行读/写操作，则不同的读/写顺序将产生矛盾的结果。

(2) 资源的最优分配对操作系统而言是困难的。例如，进程 A 申请特定的 I/O 通道，并在使用该通道前被挂起，如果操作系统封锁了该通道，并禁止其他进程使用，则将降低系统效率。

(3) 由于运行结果通常不可再现，故使程序错误的标记变得困难。

上述困难在多处理机系统中同样存在，因为这里也有进程执行的相对速度不可预测的问题，多处理机系统还必须处理多个进程同时执行带来的其他问题。例如：

```
procedure echo;
var out,in:character;
begin
    input (in, keyboard);
    out:=in;
    output (out, keyboard);
end.
```

这个过程是一个字符回显程序的基本部分，每次敲击键盘得到输入，每一输入字符都先存储在变量 in 中，再将它传给变量 out，并输出出来，任何程序都可重复调用这个过程来得到输入，并将其输出到显示器上。

对于一个支持单用户的单处理机多道程序系统，用户可以从一个应用转到另一个应用，每一应用都使用同一键盘进行输入，用同一显示器进行输出。因为每一应用都必须使用过程 echo，将其存入所有应用的全局共享存储区作为共享进程是有益的，这样只保留了 echo 的一个副本，从而节省了空间。

考虑下面的执行结果：

(1) 进程 P_1 调用过程 echo,在得到函数 input 的结果后被中断,此时输入的字符 x 存储在变量 in 中。

(2) 进程 P_2 调用过程 echo,它执行输入并在显示器上输出字符 y。

(3) 进程 P_1 再次执行,这时 x 由于被覆盖而丢失,in 中保存的是 y,y 将传给 out 而输出。

显然第一个字符丢失而第二个字符显示了两次。问题的实质在于共享变量 in 可被多个进程访问。如果一个进程修改全局变量,然后被中断,另一进程可能在第一个进程使用此变量值前被中断,假定 echo 仍是一个全局过程,只要一次只有一个进程执行该过程,则上述的执行结果就发生变化:

(1) 进程 P_1 调用过程 echo,在得到函数 input 的结果后被中断,此时,最先输入的字符 x 存储在变量 in 中。

(2) 进程 P_2 调用过程 echo,因为 P_1 仍在使用过程 echo,尽管 P_1 被挂起,但 P_2 仍被封锁在过程 echo 之外,所以 P_2 被挂起直到得到过程 echo 的使用权为止。

(3) 随后,进程 P_1 再次执行并完成 echo 的执行,字符 x 被输出。

(4) 当 P_1 退出 echo 后,解除对 P_2 的封锁,当 P_2 再次执行时,过程 echo 将被成功地调用。

这个例子说明,必须保护共享的全局变量和其他共享的全局资源,这也是控制代码访问共享变量的唯一方法。如果加上以下规则,一次只能有一个进程进入 echo,并且一旦进入 echo,其他进程就不能访问 echo,直到该过程完成为止,那么可以避免出现上述类型的错误。如何加上这种规则是本章讨论的主要问题。

这里假定问题存在于单处理机多道程序的操作系统中,以上例子说明在单处理机系统中也会出现并发问题。在多处理机系统中同样存在共享资源的保护问题,也需要同样的解决方法,首先假设对于共享的全局变量没有任何控制的访问机制。

(1) 进程 P_1 和 P_2 各自在不同的处理机上执行,并且都调用过程 echo。

(2) 以下事件以相同次序并行发生。

```
//进程 P1                    //进程 P2
    ⋮                           ⋮
input(in, keyboard)
                               input(in, keyboard)
out:=in                        out:=in
output(out, keyboard)
    ⋮                          output(out, keyboard)
                                    ⋮
```

结果是输入到 P_1 中的字符在显示前丢失,而输入到 P_2 中的字符被 P_1 和 P_2 输出。如果允许一次只能有一个进程进入 echo 的规则,则会出现以下结果:

(1) 进程 P_1 和 P_2 各自在不同的处理机上执行,P_1 调用过程 echo。

(2) 当 P_1 在执行过程 echo 时,P_2 调用 echo。因为 P_1 还在 echo 中(不管 P_1 挂起还是在执行),P_2 将被封锁进入 echo,所以 P_2 被挂起直到得到过程 echo 的访问权。

(3) 随后,进程 P_1 完成 echo 并退出。一旦 P_1 退出 echo,P_2 就再次激活并执行 echo。

在单处理机系统中,出现这个问题的原因是,进程中的中断可在任何地方中止指令的执行,在多处理机系统中除了有这个因素外,两个进程的同时执行并试图访问同一全局变量也

将引起这个问题。但是，解决这两种类型问题的方法是相同的，即有效控制对共享资源的访问。

并发的存在对操作系统的设计和管理提出了以下要求。

(1) 操作系统必须能跟踪大量活跃进程(这可使用进程控制块完成)。

(2) 操作系统必须为每一活跃进程分配资源，这些资源包括处理机时间、内存、文件、I/O 设备等。

(3) 操作系统必须保护每一进程的数据和物理资源不被其他进程侵犯，这包括与存储器、文件和 I/O 设备相关的技术。

(4) 进程执行的结果与其他并发进程执行时的相对速度无关。

3.1.1 进程间的相互作用

并发进程存在于多道程序环境，多道程序包括多个独立的应用、多进程应用和操作系统中多进程结构的使用。本节对于所述种种可能的情况，讨论进程间相互作用的方法。

根据进程觉察到其他进程的存在程度可将进程间的相互作用分为三类，如表 3.1 所示。

表 3.1 进程间相互作用

觉察程度	相互关系	一进程对其他进程的影响	潜在的控制问题
进程互不觉察	竞争	• 一进程独立于其他进程 • 进程执行时间将受影响	• 互斥 • 死锁 • 饥饿
进程间接觉察	通过共享合作	• 一进程依赖于从其他进程得到信息 • 进程执行时间将受影响	• 互斥 • 死锁 • 饥饿 • 数据一致性
进程直接觉察	通过通信合作	• 一进程依赖于从其他进程得到信息 • 进程执行时间将受影响	• 死锁 • 饥饿

1. 进程互不觉察

这些独立的进程并不协同工作。最好的例子是多个独立进程的多道程序，它们可以是大量的作业，或是相互作用的段或是二者的混合。尽管进程并不协同工作，但操作系统还是要解决竞争资源的问题。例如，两个独立的应用可能要访问同一文件或打印机，操作系统必须处理好这些访问。

2. 进程间接觉察

这些进程并不是通过名字觉察到对方，而是通过共享访问，如一个 I/O 缓冲区。这些进程通过共享方式进行合作。

3. 进程直接觉察

这些进程可通过名字直接通信并设计紧密的协同工作方式。

需要指出的是，实际情况并不像表 3.1 中所示的那样可以截然区分，一些进程可能同时表现出竞争和协作两方面，而且分别检查上述三种情况，并且在操作系统中加以实现的效率较高。

3.1.2 进程间的相互竞争

并发进程在竞争使用同一资源时将产生冲突。这种情况可简单地描述为,两个或更多的进程在执行时要访问某一资源,每一进程都没有觉察到其他进程的存在,并且每一进程也不受其他进程执行的影响。这就要求进程留下其所用资源的状态,这些资源包括I/O设备、存储器、处理机时间和时钟。

进程在相互竞争资源时并不交换信息,但是,一个进程的执行可能影响到同其竞争的进程,特别是如果两个进程要访问同一资源,那么一个进程可通过操作系统分配到该资源,而另一个将不得不等待。在极端的情况下,被阻塞的进程将永远得不到访问权,从而不能成功地终止。

进程间的竞争面临以下三个控制问题。

1) 互斥

假设两个或两个以上进程要访问同一个不可共享的资源(如打印机),那么在执行期间,每一进程将发命令给I/O设备,然后收到状态信息,发送数据并/或接收数据。这种资源称为临界资源(critical resource),访问临界资源的那一部分程序称为程序的临界段(critical section)。在任一时刻,只允许一个程序在其临界段中是至关重要的。互斥(mutual exclusion)就是要保证临界资源在某一时刻只被一个进程访问。这里不能简单地依赖操作系统来理解并执行这种约束,因为具体的需求并不是显而易见的。例如,打印机在打印整个文档时,需要有独立的进程来控制打印机,要不然就要插入相互竞争的其他进程的文档,以致出现夹杂不清的状况。

2) 死锁

互斥的实现产生两个另外的控制问题,其中一个是死锁(deadlock),即当若干进程竞争使用资源时,可能每个进程要求的资源都已被另一个进程占用,于是也就没有一个进程能继续运行,这种情况就称为死锁。考虑两个进程 P_1、P_2 和两个临界资源 R_1、R_2,假设每个进程都要访问这两个资源,这就可能出现以下情况。操作系统把 R_1 分配给 P_2,R_2 分配给 P_1,每一进程都在等待另一资源,每一进程都不释放它所占有的资源,除非它已得到另一资源并执行完临界段,故最终所有的进程将死锁。

3) 饥饿

实现互斥所产生的另一个控制问题是饥饿。假设有三个进程 P_1、P_2 和 P_3,每一个都要周期地访问资源 R。考虑以下情况,P_1 占有资源,P_2 和 P_3 都被延迟、等待资源,当 P_1 离开临界段时,P_2 和 P_3 都可分配到 R,假设 P_3 得到 R,在 P_3 完成其临界段之前,P_1 又申请得到 R,并且如果 P_1 和 P_3 重复得到 R,则 P_2 就不可避免地得不到该资源,尽管这里并没有死锁,这时称 P_2 处于饥饿(starvation)状态。

竞争的控制不可避免地涉及操作系统,因为是操作系统分配资源。另外,进程自身也必须能以某种方式表达互斥的要求,例如在使用某一资源之前先将其加锁。任何方法都要得到操作系统的支持,例如提供加锁的工具。图3.2使用抽象语言描述了互斥机制。有几个进程互斥执行,每一进程包含一个使用资源 R 的临界段和一个与 R 无关的剩余段。为实现互斥提供了两个函数:entercritical 和 exitcritical。每一函数可看作对可竞争资源名字的声明,在有某一进程进入其临界段时,任何为竞争同一资源而需要进入临界段的其他进程都将

等待，从而实现了互斥。例如：

```
program mutualexclusion;
const n=N; (* number of processor *);
procedure P(i:integer);
begin
  repeat
    entercritical(R);
    <critical section>;
    exitcritical(R);
    <remainder>
  forever
end;
begin (* main program *)
  parbegin
    P(1);
    P(2)
    ⋮
    P(n)
  parend
end.
```

3.1.3 进程间的相互合作

1. 通过共享合作

进程间通过共享合作并不需要清楚地觉察到对方。例如，多个进程可以访问共享变量、共享文件或数据库，进程不需要参照其他进程就可使用和修改共享数据，但要知道其他进程也可访问同一数据，因此，进程必须合作以管理好共享数据。这种控制机制必须保证共享数据的完整性。为此，系统中的资源应不允许用户直接使用，而应由系统统一分配和管理。

因为数据存于资源（设备、存储器）上，所以出现了互斥、死锁、饥饿等控制问题，唯一的不同是，数据项有两种不同的访问方式，即读和写，并且只有写操作必须互斥执行。

除上述问题之外还有一个新的要求，即数据相关性。例如，假设有两个数据项 a 和 b。它们要保持关系 $a=b$。也就是说，任何程序在修改一个值时也必须修改另一个以保持这种关系。现在有以下两个进程：

$$P_1: a:=a+1; \quad b:=b+1; \qquad P_2: b:=2*b; \quad a:=2*a;$$

如果起始数据是一致的，则每一进程分别执行后，共享数据仍是一致的。考虑以下两个进程在各个互斥数据上并发执行的情况：

$$a:=a+1;$$
$$b:=2*b;$$
$$b:=b+1;$$
$$a:=2*a;$$

以上执行序列结束后，就不再有 $a=b$。这个问题可通过将每个进程的整个序列声明为临界段加以解决，但严格地说这里并未涉及临界资源。

因此，可以看出在共享合作中临界段概念的重要性，这里同样可使用 3.1.2 节例子中抽象函数 entercritical 和 exitcritical。如果用临界段解决数据的完整性，那就没必要声明任何

特殊的资源或变量,在这种情况下,可以认为在并发进程中共享的声明都标记了必须互斥的临界段。

2. 通过通信合作

在以上讨论的两个例子中,每一进程都是在不含其他进程的孤立环境中。进程间的工作影响是间接的。在两个例子中用到的都是共享,在竞争的例子中,它们在没有觉察其他进程的情况下共享资源。在第二个例子中,它们共享变量,并且不能清楚地觉察到其他进程,但要保持数据的完整性。为了尽快地完成一个任务,在可能的情况下,往往将它分成若干并发执行的进程。这些进程之间必然共享一定的数据信息,必有一定的逻辑联系,这些进程即为合作进程。为了正确、有效地完成共同的任务,这些合作进程需要协同操作,它们之间也存在着直接的相互制约关系。这种直接的相互制约关系必然导致进程之间按一定的方式进行信息传递,这就是进程通信的关系。

通常,通信可描述为包括某种形式的消息,收、发信息的原语,它们是作为程序设计语言的一部分或由操作系统的内核提供。进程通信是指进程之间可直接以较高的效率传递较多数据的信息交换方式。这种方式中采用的是通信机构,在进程通信时往往以消息形式传递信息。

因为在消息传递中不存在共享,所以这种形式的合作不需要互斥,但是还存在死锁和饥饿问题。举一死锁的例子,两个进程可能由于都在等待对方的通信而阻塞。再举个饥饿的例子,假设有三个进程 P_1、P_2 和 P_3,P_1 不断地同 P_2 或 P_3 通信,P_2 和 P_3 也要同 P_1 通信,可能会出现 P_1 和 P_2 重复地交换信息,而 P_3 因为等待同 P_1 通信而阻塞,这里没有死锁,因为 P_1 保持活跃,但 P_3 却饥饿。

3.1.4 互斥的要求

并发进程的成功完成需要有定义临界段和实现互斥的能力,这是任何并发进程方案的基础,任何支持互斥的工具和方法都要满足以下的要求:

(1) 实现互斥。即在任一时刻,在含有竞争同一共享资源的临界段的进程中,只能有一个进程进入自己的临界段。

(2) 执行非临界段的进程不能受到其他进程的干扰。

(3) 申请进入临界段的进程不能被无限期延迟,也不允许存在死锁和饥饿。

(4) 当没有进程进入临界段时,任何申请进入临界段的进程必须允许无延迟地进入。

(5) 没有进程相对速度和数目的假设。

(6) 进程进入临界段中的时间有限。

这里举出三种实现互斥要求的方法。第一种方法是对并发进程不提供任何支持,因此,无论它们是系统程序或应用程序,进程都要同其他进程合作以解决互斥,它们从程序设计语言和操作系统得不到任何支持,这些称为软件方法。尽管软件方法容易引起较高的进程负荷和较多的错误,但它可使我们对并发的复杂性有较深刻的理解。第二种方法是使用特殊的机器指令,这可以降低开销。第三种方法是在操作系统中支持分级。下面就讨论这些方法。

3.2 互斥——用软件方法实现

软件方法可在共享主存的单处理机或多处理机系统中实现并发进程，这些方法通常假定基于内存访问级别的一些基本的互斥，即对内存中同一位置的同时访问（读/写）将被排序，而访问的顺序并不预先指定，除此之外不需要硬件、操作系统和程序设计语言的任何支持。

3.2.1 Dekker 算法

1. 第 1 种途径

Dekker 算法的优点在于它描述了并发进程发展过程中遇到的大部分共同问题。任何互斥都必须依赖于一些硬件上的基本约束，其中最基本的约束是任一时刻只能有一个进程访问内存中某一位置。下面以一个“小屋协议”为例来描述这个问题。

这个小屋本身和它的入口都非常小，以致在给定的任何时刻，只有一个人可以待在屋内，屋内有一块只能写一个号码的小黑板。

协议内容为，一个希望进入临界段的进程（P_0 或 P_1），进入小屋并查看黑板。若黑板上写着自己的编号，就离开小屋并去执行它的临界段；若黑板上写着他人的编号，它就离开小屋并等待，然后不时地进屋查看黑板，直至允许进入自己的临界段，这种做法称为“忙-等”（busy-waiting）。因为等待进程在进入临界段之前没做任何有意义的工作，实际上它必须不断地“闲逛”并查看小黑板，故在等待进入临界段期间，它浪费了大量的处理机时间。

在进程进入其临界段并执行完临界段后，它必须返回小屋并在黑板上写上其他进程的编号。

如果我们将这一问题用程序形式描述，则可写成如下进程形式：

```
var turn: 0..1; {turn 为共享的全局变量}
```

两个进程的程序如下：

```
//PROCESS 0                    //PROCESS 1
    ⋮                            ⋮
while turn≠0 do {nothing}    while turn≠1 do {nothing};
<critical section>;          <critical section>;
turn:=1;                     turn:=0;
    ⋮                            ⋮
```

这种方法保证了互斥，但它也有两点不足：首先，进程必须严格地交替执行它们的临界段，因此，执行的速度由进程中较慢的一个决定，如果 P_0 每小时只进入临界段一次，而 P_1 每小时进入临界段 1000 次，则该协议将迫使 P_1 的工作节拍同 P_0 保持一致；一个更严重的问题是，如果一个进程发生意外（如在去小屋的途中发生了意外），则另一个进程会永远死锁，不管进程是否在其临界段中发生意外都将导致这种情况发生。

2. 第 2 种途径

第 1 种途径的主要不足在于它只记住了允许哪个进程进入其临界段，但未记住每个进程的状态。事实上，每个进程都有进入临界段的钥匙，这样，当其中一个发生意外后，另一个仍可进入自己的临界段；每一个进程都有自己的小屋，每一个进程都可以查看另一个的黑

板,但不能修改。当一个进程想进入其临界段时,它将不时地查看另一个进程的黑板,直到发现上面写着 false 为止,这就意味着另一进程并不在其临界段内,于是它迅速进入自己的小屋,将黑板上的 false 改为 true,进程现在就可以执行其临界段了,当进程执行完自己的临界段后,再将自己黑板上的 true 改为 false。

现在,共享的全局变量是:

```
var flag: array[0..1]of boolean;
```

它被初始化为 false,两个进程的程序如下:

```
//PROCESS 0                    //PROCESS 1
    ⋮                            ⋮
while flag[1]do {nothing};     while flag[0]do {nothing};
flag[0]:=true;                 flag[1]:=true;
<critical section>;            <critical section>;
flag[0]:=false;                flag[1]:=false;
    ⋮                            ⋮
```

这时,如果进程在临界段之外,包括置 flag 的代码发生意外,那么其他进程将不会被阻塞。事实上,其他进程只要想进入自己的临界段就可以进。因为另一进程的标志 flag 永远是 false。但是,如果进程在其临界段中发生意外,或在刚把 flag 置为 true、还未进入其临界段之前发生意外,那么其他进程将被永远地阻塞。

从某种意义上说,这种解决方法比第一种更差,因为它甚至不能保证互斥。考虑如下序列:

① P_0 进入 while 语句并发现 flag[1]=false。

② P_1 进入 while 语句并发现 flag[0]=false。

③ P_0 置 flag[0]为 true 并进入临界段。

④ P_1 置 flag[1]为 true 并进入临界段。

此时,所有进程都在它们的临界段中,程序发生错误。出现这个问题的原因在于,这种解决方法依赖于进程执行的相对速度。

3. 第 3 种途径

因为一个进程可以在其他进程检查状态后,且其他进程还未进入临界段之前,可以改变自己的状态,所以第 2 种途径失败了,也许可以通过交换两行代码的次序来解决这个问题。例如:

```
//PROCESS 0                    //PROCESS 1
    ⋮                            ⋮
flag[0]:=true;                 flag[1]:=true;
while flag[1] do {nothing};    while flag[0] do {nothing};
<critical section>;            <critical section>;
flag[0]:=false;                flag[1]:=false;
    ⋮                            ⋮
```

如果一个进程在其临界段内,包括置 flag 的代码发生意外,则其他进程被阻塞,如果一个进程在其临界段外发生意外,则其他进程不会被阻塞。

先从进程 P_0 的角度来看看这种方法能否保证互斥,一旦 P_0 将 flag[0]置为 true,P_1 就不能进入自己的临界段直到 P_0 进入然后离开临界段。在 P_0 置 flag 时,P_1 可能已进入自己的临界段。在这种情况下,P_0 将被阻塞在 while 语句上,直到 P_1 离开临界段。从 P_1

的角度也可进行同样的推理。

这种方法保证了互斥但产生了另一个问题，如果所有的进程都置 flag 为 true，它们都认为对方已进入临界段，这就导致了死锁。

4. 第 4 种途径

在第 3 种途径中，一个进程在置自己的状态时，并不考虑其他进程的状态。由于每个进程都坚持要进入临界段，所以就出现了死锁。我们可以通过下面介绍的方法来解决这一问题。进程置自己的标志 flag 以表明想进入临界段，但它要时刻准备着按其他进程的状态来重置自己的标志 flag。

```
//PROCESS 0                    //PROCESS 1
    ⋮                            ⋮
flag[0]:=true;                 flag[1]:=true;
while flag[1] do {nothing};    while flag[0] do {nothing};
begin                          begin
  flag[0]:=false;                flag[1]:=false;
  <delay for a short time>;      <delay for a short time>;
  flag[0]:=true;                 flag[1]:=true;
end;                           end;
<critical section>;            <critical section>;
flag[0]:=false;                flag[1]:=false;
⋮                                  ⋮
```

这种方法已经很接近正确解法，但仍有小缺陷，采用同第 3 种途径中相同的推理可证明它能保证互斥，然而在如下的事件序列中仍会产生问题：

```
P0 置 flag[0]为 true          P1 置 flag[1]为 true
P0 检查 flag[1]               P1 检查 flag[0]
P0 置 flag[0]为 false         P1 置 flag[1]为 false
P0 置 flag[0]为 true          P1 置 flag[1]为 true
```

这个序列可以一直扩展下去，并且任意一个进程都不能进入临界段。严格地说，这并不是死锁，因为两个进程执行的相对速度的任何改变都将打破循环，并将允许其中一个进入临界段，尽管这种序列不可能保持很长，但它毕竟是一种可能的情况，因此，放弃第 4 种途径。

5. 一个正确的解决方法

通过数组 flag 可以观察到所有进程的状态，但这还是不够的，故应采取措施解决以上由于“互相礼让”产生的问题，第一种途径中的变量 turn 可用于此目的，这个变量可以指出哪个进程坚持进入临界段。

设计一个“指示”小屋，小屋内的黑板标明 turn，当 P_0 想进入其临界段时，置自己的 flag 为 true，这时它去查看 P_1 的 flag，如果是 false，则 P_0 就立即进入自己的临界段；反之，P_0 去查看“指示”小屋，如果 turn＝0，那么它知道自己应该坚持并不时去查看 P_1 的小屋，P_1 将觉察到它应该放弃并在自己的黑板上写上 false，以允许 P_0 继续执行。P_0 执行完临界段后，它将 flag 置为 false 以释放临界段，并且将 turn 置为 1，将进入权交给 P_1。

下面给出了按上述思想设计的 Dekker 算法，其证明留作练习。

```
var flag: array[0..1]of boolean;
  turn: 0..1;
procedure P0;
begin
```

```
    repeat
      flag[0]:=true;
      while flag[1] do if turn=1 then
              begin
              flag[0]:=false;
                while turn=1 do {nothing};
              flag[0]:=true;
              end;
      <critical section>;
      turn:=1;
      flag[0]:=false;
      <remainder>
    forever
  end;
  procedure P1;
  begin
  repeat
  flag[1]:=true;
  while flag[0] do if turn=0 then
         begin
           flag[1]:=false;
             while turn=0 do {nothing};
           flag[1]:=true;
         end;
      <critical section>;
    turn:=0;
    flag[1]:=false;
      <remainder>
    forever
  end;
  begin
    flag[0]:=false;
    flag[1]:=false;
    turn:=1;
    parbegin
    P0; P1
    parend
  end.
```

3.2.2 Peterson 算法

Dekker 算法可以解决互斥问题，但是，其复杂的程序难以理解，其正确性难以证明。Peterson 给出了一个简单的方法。和上述方法一样，数组 flag 表示互斥执行的每一进程，全局变量 turn 用于解决同时产生的冲突，下面给出了两进程的 Peterson 算法：

```
  var flag: array[0..1] of boolean;
    turn: 0..1;
  procedure P0;
  begin
    repeat
        flag[0]:=true;
        turn:=1;
        while flag[1] and turn=1 do {nothing};
```

```
        <critical section>;
        flag[0]:=false;
        <remainder>
    forever
end;
procedure P1;
begin
  repeat
flag[1]:=true;
    turn:=0;
    while flag[0] and turn=0 do {nothing};
    <critical section>;
  flag[1]:=false;
    <remainder>
  forever
end;
begin
  flag[0]:=false;
  flag[1]:=false;
  turn:=1;
  parbegin
    P0; P1
  parend
end.
```

容易证明这种方法保证了互斥。对于进程 P_0，一旦它置 flag[0]为 true，P_1 就不能进入其临界段。如果 P_1 已经在其临界段中，那么 flag[1]=true 并且 P_0 被阻塞进入临界段。另一方面，它也防止了相互阻塞，假设 P_0 阻塞于 while 循环，这意味着 flag[1]为 true，而且 true=1，当 flag[1]为 false 或 turn 为 0 时，P_0 就可进入自己的临界段了。下面是三种极端的情况：

(1) P_1 不想进入其临界段。这是不可能的，因为这意味着 flag[1]=true。

(2) P_1 等待进入临界段。这也是不可能的，因为若 turn=1，P_1 就能进入其临界段。

(3) P_1 反复访问临界段，从而形成对临界段的垄断。这不可能发生，因为 P_1 在进入临界段前通过将 turn 置为 0，把机会让给了 P_0。

这是一个两进程互斥的简单解决方法，进一步可将 Peterson 算法推广到多个进程的情况。

3.3 互斥——用硬件方法解决

完全利用软件方法来解决进程互斥进入临界区的问题有一定的难度，且有很大局限性，因而现在已很少采用此方法。现在有许多计算机提供了一些可以解决临界区问题的特殊的硬件指令。

3.3.1 禁止中断

在单处理机中并发进程不能重叠执行，它们只能被插入，而且进程将继续执行直到它调用操作系统服务或被中断，所以，为保证互斥，禁止进程被中断就已足够。通过操作系统内

核定义的禁止和允许中断的原语就可获得这种能力,进程可按如下方式实现互斥。

```
repeat
    <disable interrupts>;
    <critical section>;
    <enable interrupts>;
    <remainder>
forever
```

因为临界段不能被中断,互斥就得到保证。这种方法的代价较高,而且执行效率也会显著地降低,因为处理机受到不能插入的限制。第2个问题是这种方法不能用于多处理机系统。对于含有不止一个处理机的计算机系统,在同一时间通常有一个以上的进程在执行。在这种情况下,禁止中断也不能保证互斥。

3.3.2 使用机器指令

1. 特殊的机器指令

在多处理机系统中,多个处理机共享一个主存,这里并没有主/从关系,也没有实现互斥的中断机制。

在硬件水平上,对同一存储区的访问是互斥执行的。以此为基础,设计者设计了几个机器指令以执行两个原子操作,例如,读/写或读测试,因为这些操作都是在一个指令周期内完成的,所以不会被其他指令打断。

本节将给出两个最一般的指令:test and set 指令和 exchange 指令。

test and set 指令可定义如下:

```
function testset (var i:integer):boolean;
begin
  if i=0 then
    begin
      i:=1;
      testset:=true
    end
  else testset:=false
end.
```

这个指令首先测试 i 的值。如果 i 的值为0,则将 i 赋值为1,并且返回 true;反之,i 的值不变并返回 false。这里将整个 testset 函数作为一个原语执行,即这里不存在中断。

test and set 指令给出了基于这种指令的互斥协议,共享变量 bolt 初始化为0,发现 bolt 值为0的唯一进程可进入其临界段,所以其他想进入临界段的进程进入忙-等状态。当一个进程离开其临界段时,它重置 bolt 为0。此时,处于等待状态的唯一的一个进程得到临界段的访问权。刚好紧接着执行 testset 指令的进程将被选中执行。

exchange 指令定义如下:

```
procedure exchange (var r: register; var m: memory);
var temp;
begin
    temp:=m;
    m:=r;
    r:=temp
end.
```

可以用这个指令交换寄存器和内存的内容。在此指令执行期间，任何访问同一内存地址的指令将被阻塞。

下面的 exchange 指令示例给出了基于这个指令的互斥协议，共享变量 bolt 初始化为 0，发现 bolt 为 0 的唯一进程进入其临界段，它通过将 bolt 置为 1 排斥所有其他进程访问临界段。当一个进程离开临界段时，它重置 bolt 为 0，以使其他进程获得临界段的访问权。

```
test and set 指令示例:
program mutualexclusion;
const n=…; (* number of processes *);
var bolt: integer;
procedure P(i:integer);
begin
  repeat
    repeat {nothing} until testset
    (bolt);
    <critical section>;
    bolt:=0;
    <remainder>
  forever
end;
begin (* main program *)
  bolt:=0;
  parbegin
    P(1);
    P(2);
    ⋮
    P(n);
  parend
end.
```

```
exchange 指令示例:
program mutualexclusion;
const n=…; (* number of processes *);
var bolt: integer;
procedure P(i:integer);
var key_i: integer;
begin
  repeat
key_i:=1;
repeat exchange (key_i, bolt) until key_i=0;
    <critical section>;
    exchange (key_i, bolt);
    <remainder>
  forever
end;
begin (* main program *)
  bolt:=0;
parbegin
    P(1);
    P(2);
    ⋮
    P(n);
  parend
end.
```

2. 机器指令方法的特性

使用特殊的机器指令实现互斥有许多优点：

(1) 可用于含有任意数量进程的单处理机或共享主存的多处理机。

(2) 比较简单，所以易于验证。

(3) 可支持多个临界段，每个临界段用各自的变量加以定义。

下面是这类方法的一些缺陷：

(1) 由于采用忙-等，所以在进程等待进入临界段时，将耗费处理机时间。

(2) 有可能产生饥饿。当一个进程离开临界段并且有多个进程等待时，等待进程的选择是随意的，因此，不可避免地出现某些进程得不到访问权的情况。

(3) 有可能产生死锁。在一个单处理机系统中，进程 P_1 执行特殊指令(如 testset、exchange)并进入其临界段，这时拥有更高优先级的 P_2 执行并中断 P_1。如果 P_2 又要使用 P_1 占用的资源，那么互斥机制将拒绝该要求，这样它就陷入忙-等循环，但 P_1 也永远不能终止，因为它比另一预备进程 P_2 的优先级低。

由于软件和硬件方法都存在缺陷，故需寻找其他机制。

3.4 信 号 量

现在寄希望于操作系统和程序设计语言提供对并发的支持。本节开始讨论信号量,后两节介绍管程和消息传递。

1965年,Dijkstra在关于并发进程的论文中论述了解决并发进程的主要好处,提出了一种卓有成效的进程同步工具——信号量机制。Dijkstra将操作系统看作并发进程集,并描述了支持合作的可靠机制,如果处理机或操作系统提供这些机制,那么用户就能容易地使用它们。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。现在被广泛地应用于单处理机和多处理机系统以及计算机网络中。

信号量机制基本的原则:两个或更多的进程可通过单一的信号量(semaphore)展开合作,即进程在某一特定的地方停止执行直到得到一个特定的信号量。通过信号量,任何复杂的合作要求都可被满足。为了使用信号量 s 发送一个信号,进程要执行原语 signal(s)。为了得到一个信号,进程要执行原语 wait(s)。如果没有得到相应的信号,那么进程将被挂起直到得到信号为止。

信号量是一种特殊的变量,它的表现形式是一个整型变量及相应的队列;除了设置初值外,对信号量只能施加特殊的操作:P操作和V操作,P操作和V操作都是不可分割的原子操作,因此也称为原语。P操作和V操作也可记为wait()和signal(),或者down()和up()。为加深印象,可将信号量看作一个整型变量,并且定义以下三种操作:

(1) 信号量的初始化值非负。

(2) wait操作减小信号量的值,如果信号量的值为负,则执行wait操作的进程被阻塞。

(3) signal操作增加信号量的值,如果信号量的值不为正数时,则被wait操作阻塞的进程此时可以解除阻塞。

除了以上三种操作,再没有别的途径可以查看或操作信号量了。

下面给出较为正式的信号量原语定义。wait和signal都是原子操作,它们不能被中断。下面给出了二元信号量(binary semaphore)的定义,它的定义比一般信号量的定义更加严格。一个二元信号量仅能取值为0或1。二元信号量的实现更为简单,而且可以证明它与一般信号量具有同等的表达能力。

```
type semaphore=record
                 count:integer;
                 queue: list of process
               end;
var s: semaphore;
wait(s):
    s.count:=s.count - 1;
    if s.count<0
    then begin
        将该进程置入 s.queue 中;
        阻塞该进程
    end;
```

```
signal(s):
    s.count:=s.count+1;
    if s.count≤0
    then begin
      将进程 P 从 s.queue 中移出;
      将进程 P 置入就绪队列中
    end;

type binary semaphore=record
                        value: (0,1);
                        queue: list of process
                      end;
var s:binary semaphore;
waitB(s):
  if s. value=1
    then
      s.value=0
    else begin
        将该进程置入 s.queue 中;
        阻塞该进程
        end;
signalB(s):
    if s. queue 为空
    then
      s.value:=1
    else begin
        将进程 P 从 s.queue 中移出;
        将进程 P 置入就绪队列中
        end;
```

不论是采用一般信号量还是二元信号量，进程都将排队等候信号量，但这并不意味着进程移出的顺序与队列次序相同。一个最公平的规则是先进先出(FIFO)，阻塞时间最长的进程将最先从等待队列中移出。唯一的要求是进程不能在信号量队列中无限等待。

3.4.1 用信号量解决互斥问题

利用信号量可以方便地实现互斥临界区的管理要求。下面给出了一种使用信号量解决互斥问题的简明方法(与 3.1.2 节例子相比)。设用 $P(i)$标识 n 个进程，wait(s)只在其临界段前执行。如果 s 的值为负数，则该进程被挂起。如果 s 的值为 1，则其值减为 0 并且进程立即进入临界段，这时 s 的值不再为正数，所以其他进程就不能进入临界段。可以看出，一个进程要访问共享变量或共享资源(临界资源)时，可利用信号量保证正确。

```
program mutualexclusion;
const n=…; ( * number of processes * );
var s:semaphore (:=1);
procedure P(i:integer);
begin
  repeat
  wait(s);
  <critical section>;
  signal(s);
```

```
    <remainder>
    forever
end;
begin (* main program *)
    parbegin
      P(1);
      P(2);
       ⋮
      P(n);
    parend
end.
```

信号量的初始值为1,因此第一个进程执行完wait操作后可以立即进入其临界段,并置s的值为0。其他任何欲进入临界段的进程将被阻塞并置s的值为-1,这类进程的每一次失败尝试都会使s的值减小。当最初进入临界段的进程离开时,s的值减小,并且一个被阻塞进程从阻塞于该信号量的队列中移出。在操作系统的下一次调度时,它将进入其临界段。

使用信号量的互斥算法也可以用小屋模型来描述。除了黑板外,小屋中还有一个大冰箱。某进程进入小屋后执行wait操作将黑板上的数减1,这时如果黑板上的值非负,它就进入临界段;反之它就进入冰箱内冬眠。这时就允许另一进程进入小屋。当一个进程完成其临界段后,它进入小屋执行signal,将黑板上的值加1,这时如果黑板上的值为非正数,它就从冰箱中唤醒一个进程。

事实上,不难看出,在任一时刻,s.count的值可以解释如下。

- s.count≥0:s.count是执行wait(s)操作而不会被阻塞的进程数(这期间没有执行任何signal(s)操作)。
- s.count<0:s.count是阻塞在s.queue队列中的进程数。

3.4.2 用信号量解决生产者/消费者问题

生产者/消费者问题是并发进程中最常见的问题。该问题可描述如下:一个或更多的生产者生产出某种类型的数据(记录、字符),并把它们送入缓冲区,唯一的一个消费者一次从缓冲区中取走一个数据,系统要保证缓冲区操作不发生重叠,即在任一时刻只能有一方(生产者或消费者)访问缓冲区。下面用几种方法来描述信号量的能力和问题。

首先,假定缓冲区是无限大的线性数组,生产者和消费者函数的定义如下:

```
producer:
repeat
  produce item v;
  b[in]:=v;
  in:=in+1
    forever.
```

```
consumer:
repeat
  while in≤out do {nothing};
  w:=b[out];
  out:=out+1;
  consume item w
forever.
```

图3.2给出了缓冲区b的结构,生产者按自己的节拍生产数据并将其存入缓冲区。每做一次,缓冲区的索引in的值增加一次,消费者也以同样的方式工作,但消费者不会读空缓冲区,所以消费者在消费时要先确认缓冲区是否为空,若缓冲区为空,则消费者必须等待。

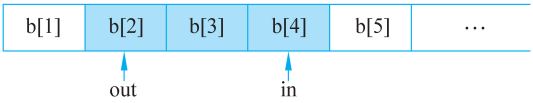


图 3.2　生产者/消费者问题的有限缓冲区

现在用二元信号量来解决此问题。下面所给的方法是首次尝试解决生产者/消费者问题，这里可以简单地用整型变量 n(＝in－out)来记录缓冲区中的数据项数，这比 in 和 out 更为简明。信号量 s 用来实现互斥。在缓冲区为空时，用信号量 delay 强迫消费者等待。

```
program producerconsumer;
var n:integer;
    s:(* binary *)semaphore (:=1);
    delay: (* binary *) semaphore (:=0);
procedure producer;
begin
  repeat
    produce;
    waitB(s);
    append;
    n:=n+1;
    if n=1 then signalB(delay);
      signalB(s)
  forever
end;
procedure consumer;
begin
  waitB(delay);
  repeat
    waitB(s);
    take;
    n:=n - 1;
    signalB(s);
    consume;
    if n=0 then waitB(delay)
  forever
end;
begin (* main program *)
  n:=0;
  parbegin
    producer; consumer
  parend
end.
```

这个方法简明易懂。在任何时候生产者都可向缓冲区中添加数据，在添加数据前，生产者执行 waitB(s)，然后执行 signalB(s)以防止在添加过程中，别的消费者或生产者访问缓冲区。在进入临界段时，生产者将增加 n 的值，如果 n＝1 则在此次添加数据前缓冲区为空，于是生产者执行 signalB(delay)，并将这个情况通知消费者。消费者最初执行 waitB(delay)来等待生产者生产出第一个数据，然后取走数据并在临界段中减小 n 的值。如果生产者总保持在消费者前面，那么消费者就不会因为信号量 delay 而阻塞，因为 n 总是正数，这样生产者和消费者都能顺利地工作。

这个方法也存在缺陷。当消费者消耗空缓冲区时，它必须重置信号量 delay，然后等待，直到生产者再往缓冲区中添加数据为止。这就是语句 if n＝0 then waitB(delay)的用途。考虑表 3.2 所示的情况，在第 6 行消费者在执行 waitB 时发生意外，消费者的确已耗尽缓冲区并置 n 为 0(见表 3.2 的第 4 行)，但是，由于生产者在第 6 行所示的消费者检测 n 之前已将 n 的值增加了，结果导致 signalB 没有与前面相对应的 waitB；第 9 行中 n 的值为－1 就意味着消费者消费了一个缓冲区中不存在的数据项，这不仅仅是改变消费者临界段中相关条件的问题，还有可能导致死锁(如第 3 行)。

表 3.2　程序中可能出现的情况

行　号	动　作	n	delay
1	初始化	0	0
2	Producer：critical section	1	1

续表

行　号	动　　作	*n*	delay
3	Consumer：waitB(delay)	1	0
4	Consumer：critical section	0	0
5	Producer：critical section	1	1
6	Consumer：if n=0 then waitB(delay)	1	1
7	Consumer：critical section	0	1
8	Consumer：if n=0 then waitB(delay)	0	0
9	Consumer：critical section	−1	0

解决这个问题的一个方法是引入一个附加变量，它设置在消费者的临界段中。这样，就不会出现死锁了。例如：

```
program producerconsumer;
var n:integer;
    s:(* binary *) semaphore (:=1);
    delay: (* binary *) semaphore (:=0);
procedure producer;
begin
  repeat
    produce;
    waitB(s);
    append;
    n:=n+1;
    if n=1 then signalB(delay);
    signalB(s)
  forever
end;
procedure consumer;
  var m:integer; (* m为局部量*)
begin
  waitB(delay);
  repeat
    waitB(s);
    take;
    n:=n-1;
    m:=n;
    signalB(s);
    consume;
    if m=0 then waitB(delay)
  forever
end;
begin (* main program *)
  n:=0;
  parbegin
    producer; consumer
  parend
end.
```

使用一般信号量可以得到另一种解决方法，变量 n 是一个信号量，它的值等于缓冲区中的数据项数。现在假设对程序进行改动，交换 signal(s)和 signal(n)的位置，这就使

signal(n)在生产者的临界段中执行而不会被消费者或其他生产者中断。上述改动并不影响程序的正确性，因为在任何情况下，消费者在执行前都必须等候所有的信号量。例如：

```
program producerconsumer;
var n: integer (:=0);
    s: semaphore (:=1);
procedure producer;
begin
  repeat
    produce;
    wait(s);
    append;
    signal(s)
    signal(n)
  forever
end;
procedure consumer;
begin
  repeat
    wait(n);
    wait(s);
    take;
    signal(s);
    consume;
  forever
end;
begin (* main program *)
  parbegin
    producer; consumer
  parend
  end.
```

如果再交换 wait(n)和 wait(s)的次序，这就会产生一个致命的错误，当缓冲区为空时(n.count=0)，如果消费者进入临界段中，那么生产者就不能向缓冲区中添加数据，从而系统陷入死锁。这个例子说明了使用信号量要注意细节以及进行正确设计的困难性。

最后，给生产者/消费者问题添加一个新的也是现实的约束，即缓冲区有限。如图 3.3 所示，将缓冲区看作一个环，并且指针的值必须对缓冲区的容量取模。

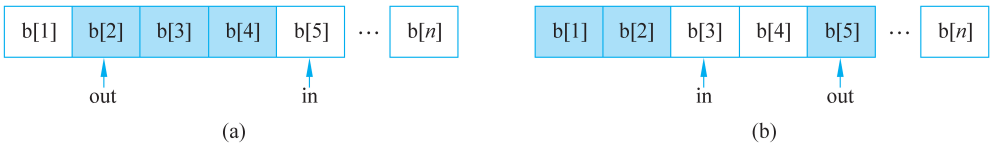


图 3.3　生产者/消费者问题的有限环形缓冲区

生产者/消费者函数定义如下(其中 in 和 out 的初始值为 0)：

```
producer
repeat
  produce item v;
    while((in+1) mod n=out) do {nothing};
      b[in]:=v;
      in=(in+1) mod n
forever;
```