# 第3章 处理器管理

处理器是计算机系统最重要的资源，计算机系统的功能是通过处理器运行程序指令来体现，计算机系统的工作方式主要是由处理器的工作方式决定，因此，处理器管理成为操作系统的核心功能。通过引入多道程序并发执行的工作方式，实现从单任务到多任务的转变。本章和第4章主要介绍单处理器系统中并发执行工作方式的管理和控制的原理和方法。

## 3.1 系统的工作流程

本节进一步介绍多道程序设计并发执行的思想，分析并发执行的复杂性。

处理器执行程序的方式称为系统工作方式，也称工作流程，系统的基本工作流程是：顺序执行和并发执行。

### 3.1.1 程序及其特点

为了更好地理解以后章节的内容，这里先补充介绍程序的概念。

程序是能够完成特定功能的一组指令(或语句)的有序集合。在结构化程序设计中，程序又往往由一个主程序和一些更小的模块或子程序组成，主程序与子程序之间、子程序与子程序之间按功能的实现要求，通过调用/返回方式建立联系。在面向对象的程序设计中，对象本身或对象中的方法也称作模块。

程序应具备以下两个基本特点。

#### 1. 顺序性

顺序性是指：处理器在执行一道程序的指令时，应严格按照程序员规定的指令顺序逐条执行。对于一道程序，它的指令顺序是程序员按照算法的要求精心设计排列的，如果处理器没有完全按照这个顺序执行，这道程序的功能肯定得不到正确的处理。所以对于每道程序，它的指令是依次地在处理器上执行，即只有一条指令完成后，才能执行它的下一条指令。

#### 2. 可再现性

一道正确的程序，只要初始条件相同，在同一台机器上多次运行，每次运行得到的结果就应该相同。或者说，一道程序，如果在不同时间多次运行，那么，每次运行所完成的功能应该是相同的。这种特点称为程序的可再现性，或称再现性，程序的可再现性有着重要的作

用，程序员调试程序的依据就是利用程序的可再现性。一般地，程序员在编写程序时，如果发现在输入的初始参数相同的情况下，同一道程序的两次运行结果不同，那么，可以断定这道程序存在错误，应该认真分析源代码，找出错误原因。

### 3.1.2 顺序执行的工作方式及特征

顺序执行是早期的计算机系统所采用的工作流程，是一种简单的处理器工作方式。顺序执行是指：处理器在开始执行一道程序后，只有在这道程序运行结束后（程序的指令全部运行完成，或程序运行过程出现错误而无法继续运行），才能开始执行下一道程序。

这种工作流程的外在表现就是单任务，DOS 操作系统就是采用这种工作方式。

顺序执行具有封闭性和可再现性两个基本特征。

#### 1. 封闭性

所谓封闭性，就是运行中的程序不受其他程序的影响。例如，运行的程序在申请资源时，不必担心所申请的资源被其他程序使用。

封闭性使得系统资源的管理简单，但全部资源由运行的程序单独使用，没有共享，所以资源利用率低。

#### 2. 可再现性

处理器的顺序执行工作方式，支持程序的可再现性特点。也就是说，正确的程序，在初始条件相同的情况下，不同时间多次运行，得到的结果相同。

所以，在顺序执行的工作方式下，处理器不需要额外的控制就可以保证程序正确地执行。

### 3.1.3 并发执行的工作方式及特征

什么是并发执行(Concurrence)？并发执行是指：在多道程序设计环境下，处理器在开始执行一道程序的第一条指令后，在这道程序完成之前，处理器可以开始执行下一道程序，同样地，更多其他的程序也可以开始运行。也就是说，处理器在执行一道程序的两条指令之间，可以执行其他程序的指令。

这种工作流程的外在表现就是多任务，现代的计算机系统都采用了这种工作方式。

多道程序的并发执行，可以从宏观和微观两个方面来理解。

宏观上，多道程序“同时”在运行，表现为多任务。因为一道程序开始后，在它完成之前，它是处于运行之中，这时另外的程序也可以开始运行，它们也在运行之中，所以多道程序同时都在运行中。

微观上，多道程序又是轮流交替地在处理器上执行。由于只有一个处理器，而一个处理器任何时刻至多只能执行一条指令，这条指令只能属于一道程序，所以在微观上任何时刻至多只有一道程序真正在运行之中。

并发执行并没有破坏程序的顺序性特点，因为在多道程序的轮流交替执行过程中，对于一道程序而言，处理器仍然按照程序的指令顺序依次逐条地执行，只是这道程序在处理器上

的执行被分割成多个时间段，表现为“停停走走”的过程。

并发执行具有两个突出的优点。从 1.2.2 节例 1-1 可以看出，多道程序的并发执行可以提高处理器的利用率，因为多任务的“同时”运行，可以发挥在硬件上处理器与设备、设备与设备并行工作的能力。另外，多道程序的并发执行也为任务协作提供了可能。

多道程序并发执行的工作流程是本章重点讨论的内容，它在宏观、微观上的思想贯穿以后的各个章节。

并发执行具有复杂性，主要体现在以下几个方面。

### 1. 随机性

为了表现多道程序并发执行在宏观上的多任务，微观上这些程序必须轮流交替地在处理器上运行，并且，在这些程序中，哪一道程序先运行、哪一道程序后运行、运行时能够连续运行多长时间等都是随机的、不确定的。

也就是说，在这些多道程序中，哪一道程序先运行都是允许的，运行时能够连续运行多长时间也可以是任意的。并发执行的随机性为系统的管理、控制带来灵活性，因为只要需要，系统就可以让任何一道程序运行。

### 2. 不可再现性

多道程序并发执行破坏了程序的可再现性。也就是说，正确的程序，在相同的初始条件下，不同时间运行，先后可能得到不同的结果，下面通过一个简单而又典型的例子来说明这个问题。

**例 3-1** 已知两道程序 PA 和 PB，它们对同一个变量 count 进行操作，PA 程序每次运行时对变量 count 进行加 1 操作，而 PB 程序每次运行时对变量 count 进行减 1 操作。用 C 语言语法描述 PA 和 PB 程序如下。

```
PA(){
  int x;
  x = count;
  x = x + 1;
  count = x;
}
PB(){
 int y;
 y = count;
 y = y - 1;
 count = y;
}
```

可以假定：变量 count 为 int 类型，在 count＝100 时 PA()和 PB()各运行一次，那么，在并发执行方式下，它们运行后 count 的值是多少？

在正常情况下，当 count＝100 时，PA()和 PB()各运行一次，结果应该还是 count＝100。但是，在多道程序并发执行方式下，将可能产生不同的运行结果。

为方便说明，为 PA()和 PB()程序中的主要语句加上编号。

PA()程序中的主要语句及编号是：

① x＝count；

② x＝x＋1；

③ count＝x；

PB()程序中的主要语句及编号是：

④ y＝count；

⑤ y＝y－1；

⑥ count＝y；

由于并发执行的随机性，PA()和 PB()在运行过程中可以有许多种的轮流交替方式。下面来分析三种可能出现的轮流产交替执行。

(1) 按①②③④⑤⑥执行

这是一种特殊的方式，其实就是先运行 PA()全部语句，完成后再运行 PB()的全部语句。容易得出，在 PA()运行完成后，count＝101，接着，在 PB()运行完成后，count＝100，这个结果与实际是相符的，是正确的。

同样可以发现：如果按④⑤⑥①②③执行，也可以得到正确的结果：count＝100。

这两种特殊的轮流交替方式其实就是按单任务顺序执行的工作流程，在顺序执行方式下可以保证程序的可再现性。

所以，在两种基本工作流程中，并发执行方式包含了顺序执行方式。

(2) 按①④⑤⑥②③执行

处理器先执行 PA()的语句①后，$x＝100$；这时处理器暂停 PA()的执行，现场保护后转而执行 PB()的④⑤⑥，在执行 PB()的④语句之前 count＝100；因此，在 PB()的④⑤⑥语句执行后，count＝99，但这并不是最终结果，接着处理器还要执行 PA()剩余语句②③，处理器在继续执行 PA()之前要恢复之前暂停时的现场(包括其中的 $x＝100$)，然后再执行语句②③，在 $x＝100$ 时，执行②③的结果是：count＝101。

这个结果不符合实际情况，是个错误的结果。

(3) 按④①②③⑤⑥执行

处理器先执行 PB()的语句④后，得 $y＝100$；这时处理器暂停 PB()的执行，转向执行 PA()的①②③，在执行 PA()的①语句之前 count＝100；因此，在 PA()的①②③语句执行完成后 count＝101，接着处理器执行 PB()剩余语句⑤⑥，处理器在继续执行 PB()之前要恢复之前暂停时的现场(其中有 $y＝100$)，然后才执行语句⑤⑥，在 $y＝100$ 时，执行⑤⑥的结果是：count＝99。

这个结果也不符合实际情况，是个错误的结果。

还有其他的轮流交替方式，留给读者自己分析。

综合上述分析，并发执行方式可能造成一种现象，表面上处理器工作正常，而实际程序运行的结果却是错误的。那么，这种错误的原因又在哪里呢？

因为 PA()和 PB()只是分别实现加 1 和减 1 的简单操作，可以看出所给的程序代码本身没有错误。

所以，在并发执行方式下，正确的程序可能得不到正确的运行结果，程序的可再现性特点被破坏了。

### 3. 相互制约

并发执行方式复杂性的另一个表现是程序之间相互制约，例如，在微观上，一道程序在运行时其他程序不能运行；一道程序在使用一些资源时，影响了其他程序对同样资源的使用。对一道程序而言，没有了之前顺序执行方式的“封闭性”。

在第 4 章将了解到，并发执行的相互制约还体现在“进程死锁”。

综上所述，多任务的并发执行具有复杂性，操作系统需要对并发执行实施严密的管理和控制，才能发挥这种工作方式的优点。

最后，简要介绍并行执行(Parallel)方式。在字面上，并行执行与并发执行很相近，但两者本质不同，这里通过例子进行说明。例如，A 和 B 两道程序，如果说 A 和 B 并发执行，是指 A 和 B 两道程序共同使用一个处理器，在微观上，它们只能轮流交替地运行。如果说 A 和 B 并行执行，是指 A 和 B 两道程序各自在不同的物理部件(处理器、设备等)上执行，例如程序 A 在一个处理器上运行，程序 B 在另一个处理器上运行，在微观上，它们可以真正同时地在不同的物理部件(处理器、设备等)上独立运行，运行时各自独立，彼此不受影响。

程序的并行执行是多处理器系统的工作方式。本书以单处理器为管理对象，所以，只考虑并发执行方式，在提到并行工作时，是指处理器与设备或设备与设备的并行工作。

## 3.2 进程的概念

### 3.2.1 进程的定义

经过 3.1.3 节的分析可以知道，系统的工作流程采用多道程序的并发执行方式后，正确的程序得不到正确的运行结果，所以，程序这个概念不能满足操作系统的要求。为了实现并发执行，分析、解决并发执行中出现的问题，操作系统引入一个新的概念，即进程，来深入揭示程序的运行规律和动态变化。

什么是进程？这里引用著名的荷兰计算机专家 Dijkstra 对进程下的定义。在 20 世纪 60 年代初，Dijkstra 参加一个支持多道程序的操作系统 THE 的设计与实现，他把一道程序在一个数据集上的一次执行过程，称为一个进程(Process)。与此同时，IBM 公司在设计开发他们的操作系统时也独立地提出：程序的运行过程称为任务(Task)。任务和进程都是对运行程序的描述，两者可以交换使用，不作区别。但是由于 Dijkstra 在进程方面相继提出许多开创性的理论成果，人们更倾向于采用 Dijkstra 对进程的定义。

进程与程序有什么联系？首先，进程是程序的运行过程，因此进程包含了程序；其次，进程的运行就是其对应的程序的运行；最后，程序规定了进程所要完成的功能。

引入进程，不仅可以实现对处理器的有效管理，同时也实现了对其他资源的管理，所以，进程是操作系统最基本的、最重要的概念，接下来的几节内容还是围绕着进程的概念展开，进一步揭示程序的运行规律和动态变化。

### 3.2.2 进程的主要特征

进程是程序的运行过程，它有 5 个特征：动态性、并发性、独立性、结构性和异步性。

#### 1. 动态性

进程的动态性是指：每个进程都有一个生命期，具有一个从创建、运行到消亡的过程。与哲学上的物质概念一样，物质是运动的，具有一个从产生、发展，再到消亡的过程。

动态性是进程的基本特征之一。关于这个方面内容在 3.3 节进一步分析。

进程与程序的根本区别就在于：进程是动态的，而程序是静态的。程序可以以纸质或电子存储介质等形式存在，如果程序员没有修改，程序还可以长期保存；进程是程序在处理器上的运行过程，是动态变化的，具有从产生到消亡的过程。

#### 2. 并发性

并发性是进程的另一个基本特征。

多个进程可以并发执行。一个进程被创建后，在它消亡之前，其他的进程也可以被创建。这样，宏观上有多个进程同时在运行中，但是对于单处理器，任一时刻至多只能运行一个进程的程序代码，因而微观上，这些进程只能是轮流交替地在处理器上运行。这种轮流交替具有随机性、不确定性，也就是说，处理器先运行哪个进程、后运行哪个进程、一个进程在处理器上运行时能够连续运行多长时间等等，都是不可预知的。

并发执行的进程也简称并发进程。

#### 3. 独立性

为了管理的方便，操作系统规定进程应该具备独立性。进程独立性具体表现在：进程是操作系统分配资源的基本单位，一个进程的程序和数据只能由该进程本身访问（也就是后续章节中所说的进程的地址空间是私有的）。

进程的独立性要求并发进程之间，在同一个处理器上运行时，各自都能够正确地完成程序所规定的功能。

#### 4. 结构性

在多道程序设计环境下有很多进程，但是它们具有相同的属性，操作系统经过概括、抽象后，定义一个相对固定的格式即数据结构，用于表示一个进程，这个数据结构就是进程控制块 PCB。

#### 5. 异步性

多个进程并发执行时，每一个进程的运行过程不可预知，因此，它何时运行完成也无法准确预知，这就要求操作系统必须做到，在一个进程运行完成之前，随时可以创建一个或更多新的进程，这就是进程的异步性。

通过上述的进程的 5 个特征，可以进一步加深对进程概念的理解。

## 3.3 进程的动态性

本节进一步讨论和分析进程的动态性特征，深入揭示程序的运行规律和变化状态。

### 3.3.1 进程的基本状态

Dijkstra 在给出进程的定义后，对进程生命期的变化状况作出了更细致的划分，把一个进程在创建后、消亡之前分为三个基本状态：运行(Running)、就绪(Ready)和阻塞(Blocked)。

#### 1. 运行状态

称一个进程处于运行状态是指：处理器当前执行的指令正是该进程对应的程序代码。经常也可以描述为：进程正占用 CPU 运行，或者说 CPU 分配给进程。

并发进程在宏观上表现为多任务同时运行，但是，这里的“同时”只是从用户角度观察、感觉到的，并不是真正的运行，只有处于运行状态的进程，才是真正在运行之中。

运行状态也称执行状态。

#### 2. 就绪状态

在单处理器系统的多道程序设计环境中，至多只有一个进程处于运行状态，其他进程暂时不能运行，即不处于运行状态。

不在运行状态的进程又分两种情况，其中之一就是就绪状态的进程：对于当前不在运行状态的进程，如果把 CPU 分配给它，它就可以立即运行，这样的进程称为处于就绪状态的进程。

可见，就绪状态的进程已经得到了除处理器外的所有资源，只是因为缺少处理器，假如有足够的处理器数量，就绪状态的进程也都可以运行。

#### 3. 阻塞状态

不在运行状态的进程，除了就绪状态的进程之外，另一种就是阻塞状态的进程。也就是说，对于当前不在运行状态一个进程，即使把处理器分配给它，它也不能运行，这样的进程称为处于阻塞状态的进程，阻塞状态也称等待状态。

把处理器分配给它，它也不能立即运行，这样的进程存在吗？实际上，正是因为阻塞状态才使得多道程序的并发执行有意义。所以，这种进程不仅存在，而且还很普遍。

程序具有顺序性，处理器执行程序，是按程序员事先设计的顺序依次地执行程序中的指令，当前的一条指令在没有执行完成之前，同一程序的下一条指令就不能开始。

一个进程在运行过程中，可能随时需要申请新的资源，由于多道程序设计环境下其他进程的存在，这个申请可能得不到满足，这样，在没有得到所需的资源之前，后续的指令不能运行，也就是说，这时，即使把处理器分配给它，它也不能向前推进。操作系统通过引入阻塞状态，把这样的进程设置为阻塞状态，将处理器让给其他可运行的进程，以减少处理器的等待

时间。

还有，对于需要与设备进行数据交换操作（即 I/O 操作）的进程，处理器在执行程序的 I/O 操作请求时，启动设备并发送 I/O 请求，接着，处理器就等待设备 I/O 操作的完成，对于相对快速的处理器而言，设备的 I/O 操作速度比较慢，这样的等待时间可能会很长。所以，处理器在启动设备 I/O 操作成功后，如果简单地让处理器等待，直到 I/O 操作完成再执行程序的后续指令，那么，也就无法发挥硬件上处理器与设备并行工作的能力，这就需要操作系统采取办法，以便发挥出硬件的这种并行能力。这个办法就是：把执行 I/O 操作的进程置为阻塞状态，处理器分配给下一个进程，实现了处理器执行的同时，设备也在进行 I/O 操作。由此看出，一个进程在提出 I/O 操作请求后，在它的 I/O 操作完成之前，把处理器分配给它，它也不能运行。

读者自己也可以经过分析，发现其他的类似情况。

因此，阻塞状态的进程是存在的。正是因为引入进程的阻塞状态，多道程序技术才可以把硬件上具有的处理器与设备、设备与设备的并行能力发挥出来。

一个进程在它的生命期内的任一时刻，一定是处在某个状态且只能在一个状态上。

### 3.3.2 基本状态的转换关系

进程有三个基本状态：就绪、运行和阻塞。进程的动态性特征说明，进程是运动变化的。正常情况下，一个进程不能永久处于一个状态，在适当的时候，需要从一个状态向另一个状态转换，这种的状态转换是怎样的？图 3-1 描述了进程生命期三个状态的转换关系。

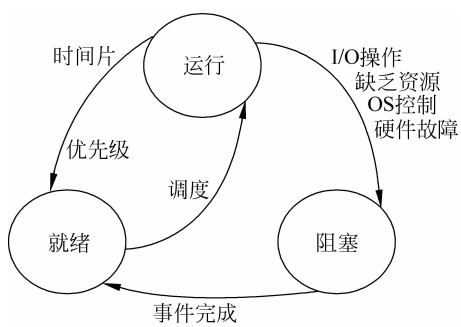



图 3-1 进程状态转换关系图

操作系统通常规定，新创建的进程其状态为就绪状态，在多道程序设计环境下，就绪状态的进程可能有多个，操作系统的调度程序根据一定的策略从中选择一个让它占用处理器运行，选中的进程就转化为运行状态。

由于并发执行的随机性，运行中的进程可能要回到就绪状态，例如，在分时系统中，当前运行进程的时间片用完而进程的任务还没有结束，这时进程就要回到就绪状态，系统把处理器分配给下一个进程运行。还有，在实时系统中，可能出现一个任务紧迫的事件需要优先处理，这样原来运行的进程也要暂时回到就绪状态，处理器转去执行任务紧迫的进程。回到就绪状态的进程，将来合适的时候，经调度程序再次选中又进入运行状态继续运行。

运行状态的进程，处理器按照进程所对应程序的指令顺序，逐条地运行，期间个别特殊指令的执行会导致进程进入阻塞状态。例如，运行的进程执行了 I/O 请求，处理器在启动

I/O操作成功后，处理器就等待I/O操作的完成，操作系统为了减少处理器的等待时间，把当前进程设置为阻塞状态，处理器分配给下一个进程，这样可以实现处理器与设备、甚至设备与设备的并行，从而达到引入多道程序设计的目的。

导致进程从“运行状态”转换到“阻塞状态”的原因，除了I/O操作请求之外，常见的还有以下几种。

### 1. 缺乏资源

运行的进程在动态申请资源得不到满足，即缺乏资源时，运行状态的进程进入阻塞状态。程序设计语言提供程序员灵活的编写代码的方法，进程可以根据需要动态地提出申请新的资源，或归还(或释放)不再使用的资源。由于多道程序的并发执行破坏了封闭性，运行进程所申请的新资源可能已经被其他进程占用，使得当前进程得不到运行所需的资源暂时无法继续运行，这时操作系统让运行进程进入阻塞状态。

### 2. 系统控制

在本章3.6节将介绍，因为并发执行的相互制约，操作系统为了控制的需要，可能强制让一个运行进程暂时停下来，否则可能导致一些错误结果。操作系统把这些需要控制的进程设置为阻塞状态。

### 3. 硬件故障

在处理器运行时出现了硬件故障，如读取内存数据时出错等，也将使得运行状态的进程转换为阻塞状态。

在阻塞状态的进程，当对应的原因解除后，转化为就绪状态。如对应的I/O操作完成，或者其他进程归还或释放了足够的资源，或者操作系统在控制一段时间后认为运行时机成熟，或者硬件故障得到排除等，相应的进程就转化为就绪状态。

经过上述状态转换的分析，可以对进程的动态性特征有更深入的认识。一个进程在创建后处于就绪状态，被调度程序选中后进入运行状态，运行过程中可能因并发执行的原因回到就绪状态，或者因为特殊操作进入阻塞状态，阻塞状态的进程在造成阻塞的原因解除后唤醒，恢复为就绪状态；就绪状态的进程由调度程序选中后继续运行，如此反复，直到所有进程运行完成。

这里，有两个问题需要进一步讨论。

“因为进程有三个基本状态——就绪、运行和阻塞，所以每个进程在其生命期内，都要经历这三个状态。”这种观点正确吗？

虽然进程有三个基本状态，但是对于每一个进程而言，在它生命期内不一定都要经历这三个状态。新创建的进程其状态为就绪状态，经调度程序选中后进入运行状态，对于一些纯计算性的简单进程，在很短时间的运行后就结束了，也就无须进入阻塞状态。所以，对于个别的进程，阻塞状态可以不要经历。

另外，阻塞状态的进程将来是转换为就绪状态，能否把阻塞状态的进程在其阻塞原因解除后，直接转换为运行状态？这个问题将在3.5.6节回答。

## 3.4 进程管理的主要功能

系统工作流程从单任务的顺序执行发展到多任务的并发执行，可以发挥硬件上处理器与设备并行工作的能力，引入进程的概念，对处理器的管理转化为对进程的管理。

### 3.4.1 进程控制块及其组成

本章前面三节的内容从原理上介绍进程的概念，重点分析了进程的动态性特征。那么操作系统作为系统软件，又是如何定义和描述进程的呢？

进程具有固定的结构形式，这就是进程的结构性特征。操作系统把为描述、管理和控制进程所设计的数据结构称为进程控制块，简称 PCB(Process Control Block)。

创建一个进程就是为其建立一个 PCB，进程状态的转换就是通过修改 PCB 中状态的值实现的，在 PCB 中还描述了进程使用资源情况等。进程运行完成后，PCB 被回收，进程也就结束。所以，PCB 是进程存在的标识，操作系统通过 PCB 管理控制进程。

PCB 是一个较为复杂的数据结构，可分为 3 个组成部分。

#### 1. 基本描述信息部分

基本描述信息部分的数据主要是描述进程信息，其中包括以下几方面。

(1) 进程名

进程的名称(pname)，通常是用程序文件名或命令名称表示。

(2) 进程标识符

进程标识符(pid)由操作系统自动生成，pid 是唯一的，可以用于区别进程。这里所说的 pid 唯一性，是指同一台计算机，在一次开机后，关机之前，期间的所有进程的 pid 各不相同。

在 UNIX 操作系统中，使用 getpid()系统调用可以获取当前进程的 pid。

(3) 用户标识

创建进程的用户标识(uid)。现代操作系统都支持多用户，在安装操作系统时有一个默认的管理员用户，例如 UNIX 操作系统的 root 用户、Windows 操作系统的 Administrator 用户，其他的用户需经管理员注册、授权。用户使用计算机时，需经过登录操作，在操作系统的身份认证后才能使用计算机。

操作系统在用户注册时，为每个用户分配一个唯一的标识符 uid。

(4) 进程状态

表示进程当前的状态(pstate)。

此外，还有其他信息，如父进程等。

#### 2. 管理信息部分

管理信息部分主要对进程运行过程中所需要的资源等信息进行登记，主要有以下几方面。

(1) 程序和数据的地址

进程对应的程序和数据的地址,与采用哪种主存储器管理方法有关,如页表起始地址、长度,或分区起始地址及长度,或分区号等。

(2) I/O 操作相关参数

在进程 I/O 操作时需要的参数,如设备逻辑号、传输的数据量大小、缓冲区地址等。

(3) 进程通信信息

进程之间通信时的相关数据,如消息缓冲队列指针等。

此外,还有其他信息,如 PCB 结点的指针信息,用于指示下一个进程的 PCB 地址。

#### 3. 控制信息部分

操作系统为了控制进程的运行,需要登记的信息有以下几种。

(1) 现场信息

进程从运行状态进入阻塞状态时,CPU 的各主要寄存器内容,如标志寄存器、堆栈寄存器、段寄存器、通用寄存器等内容需要保护,以保证下次能够接着继续运行。

(2) 调度参数

进程调度程序执行时所需的参数称为调度参数,例如,到达时间、优先级、进程大小、累计运行时间等。

(3) 同步、互斥的信号量

例如,在消息缓冲队列通信中所需要的同步、互斥的信号量。

以上介绍了 PCB 的基本组成,对于不同的操作系统产品,PCB 结构在实现上有较大的差别。

### 3.4.2 PCB 队列

系统中一个进程对应一个 PCB,而一个 PCB 也唯一对应一个进程。在多道程序设计环境中,可能同时有多个进程,对于这些进程,操作系统用队列来管理,称为进程队列或 PCB 队列。队列通常又以链表的数据结构实现,所以也称 PCB 链表。

在一个系统中,PCB 队列往往有多个,可分为两类:就绪队列和等待队列。

#### 1. 就绪队列

把就绪状态的进程的 PCB 链接起来组成的队列称为进程就绪队列,简称就绪队列,在一些操作系统中,按进程的性质或优先级不同,就绪队列还可进一步细分成多个就绪队列,这种做法的好处是:可以尽量让进程调度算法只在小范围内选择一个进程,保证算法拥有较好的性能,同时,让一些就绪进程暂时不参与竞争使用计算机系统的部分资源。

在多处理器系统中就绪队列也称为请求队列,每个处理器都对应一个请求队列,一个进程被创建时,处理器分配算法为新进程选择一个准备为它运行的处理器,并把它加入到对应的请求队列中,请求队列中的进程经过调度程序选择后才能运行。

#### 2. 等待队列

把阻塞状态的进程的 PCB 链接起来组成的队列称为进程等待队列,简称等待队列。等

待队列也有多个，按照造成阻塞的原因分类，例如，磁盘 I/O 请求的等待队列、打印机请求队列、内存申请等待队列，等等。

一个进程对应一个 PCB，多个进程通过 PCB 队列组织，在多道程序设计环境下，操作系统就是以这种方式实现进程的有效组织。

### 3.4.3 进程管理的主要功能

操作系统进程管理的主要功能是：控制、同步、通信、调度和死锁。

进程控制功能是对进程生命期及其状态转换的实现，进程同步是对并发执行进程的控制以保证程序的可再现性和任务协作，进程通信实现进程之间的数据交换，进程调度则是实现并发执行微观上的轮流交替运行，进程死锁是分析、解决并发执行的另一种错误现象。

本章后续内容将分别介绍进程管理的控制、同步和通信，第 4 章介绍进程管理的调度和死锁。

## 3.5 进程控制

操作系统是计算机系统上配置的第一个大型软件，管理、控制着计算机的工作过程。作为一个最基本的软件，有些操作的执行具有很严格的要求，其中之一就是原子操作。

本节先介绍原语的概念，然后介绍进程管理的第一个功能——进程控制。

### 3.5.1 原语

一个操作依次分成几个动作，如果这几个动作的执行满足特性：一是不会被分割或中断，二是这些动作要么全部执行，要么一个都不执行，则称这种操作为原子操作，其中满足的特性称为原子性。原子操作具有原子性(即 All or Nothing)。

什么是原语(Primitive)？一个特殊的程序段称为原语，如果这段程序的执行具有原子性，也就是这段程序的所有指令，要么全部执行了，要么一个都不执行，处理器一旦开始了第一条指令的执行，接下来只能执行这段程序的后续指令，直到完成，期间不能转去执行其他程序的代码，任何两条相邻指令的执行不可被分割或中断。

原语中的指令执行具有很高的要求，如果执行原语中某一条指令时出错了，那么该原语之前已经执行的指令要恢复到执行前的状态。

原语也称广义指令，原语中的几条指令被看成一个实体。

原语的主要作用是保证系统运行的一致性。

### 3.5.2 进程控制的含义

进程控制就是实现对进程状态的转换，这种转换是通过一组原语来实现的。进程控制原语主要有创建、撤销、阻塞、唤醒、切换等。

### 3.5.3 进程的创建

进程创建原语(Create)，用于创建一个进程，创建后，进程的生命期就开始了。

### 1. 创建进程的时机

什么时候需要创建进程？操作系统启动的过程中，由初始化程序自动创建一些系统进程，除此之外，主要有以下 3 种时机。

(1) 作业调度程序

在批处理系统中，作业调度程序从作业后备队列中选中一个作业，之后为该作业的每个作业步创建一个进程。

(2) 用户提交请求命令

用户通过键盘、鼠标等输入设备提交请求命令后，操作系统命令解释程序的进程接收这个命令，如果该命令是合法、有效的，命令解释程序的进程为用户提交的请求命令创建一个进程。

(3) 系统调用

操作系统提供创建进程的系统调用(如 UNIX 的 fork)，程序员可以根据需要，利用系统调用创建一个新的进程，新进程是原进程的子进程，原进程是新进程的父进程。从进程角度来看，子进程也是进程，与父进程没有区别，彼此是独立的，父进程、子进程之间也可以并发执行。

### 2. 创建原语的主要操作

创建原语的主要操作如下。

(1) 建立一个 PCB

如果是用链表组织进程，则新建一个 PCB 结点，如果是用进程表(Process Table)组织进程，则只需从进程表中找出一个空表项(Item)，即空行。所谓进程表，就是由行和列构成的一张二维的表，表的长度(即行数)固定。

(2) 生成 pid

系统有一个专门的标识符生成器，为每个新进程生成一个唯一的标识符 pid，这里的“唯一”只需要在同一台计算机上的一次开机和关机之间每个进程的 pid 不同即可。同一台计算机下次开机后创建的进程的 pid 允许与上次开机时的某个进程的 pid 相同，这种相同不影响系统的进程管理，因为上次开机中的进程已经不存在了。

另外，不同机器之前的进程 pid 也不需要限制。

(3) 初始化 PCB 各项内容

调用操作系统的其他功能模块，为进程分配运行所需的基本资源，初始化 PCB 的各项数据，其中，进程状态为就绪状态。

(4) 加入合适的就绪队列

最后，根据进程的性质，将新进程的 PCB 加入合适的就绪队列中。

### 3. 进程树

虽然系统中的进程以队列方式组织，但可以通过 PCB 中的相关数据，构成一个以 PCB 为结点的树，称为进程树。

操作系统启动成功后，自动建立一个系统进程，以该进程为树的根结点。作业调度程序

创建的进程、或者命令解释进程创建的进程作为根结点的子结点，运行中的进程可能根据需要创建它的子进程结点，子进程还可以创建子进程。这样，系统中的进程之间具有类似“家族”的关系，如图 3-2 所示。

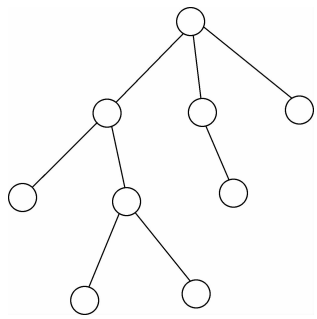


图 3-2　进程树

### 3.5.4　进程的撤销

进程撤销(Destroy)标志进程的生命期结束，即消亡。

撤销进程的时机有：进程执行完成；进程执行过程出错(也称异常结束)；子进程对应的父进程异常结束；人为操作终止进程等。

进程撤销的主要功能是“回收”资源，把进程所占用的资源回收，以供其他进程使用也包括 PCB 的回收。

### 3.5.5　进程的阻塞

运行状态的进程要进入阻塞状态时，通过阻塞(Blocked)原语实现。阻塞原语的主要功能如下。

#### 1. 修改 PCB 中的进程状态

把原来的运行状态设置为阻塞状态。

#### 2. 现场保护

将处理器现场的内容保存在 PCB 中。

#### 3. 将进程加入合适的等待队列

阻塞原语的执行将引起新的调度，因为运行进程阻塞后，处理器即将空闲，操作系统的进程调度程序选择一个进程运行。

### 3.5.6　进程的唤醒

当阻塞状态的进程要转换为就绪状态时，通过唤醒(Wakeup)原语实现。唤醒原语的主要操作如下。

### 1. 从等待队列中移出进程

需要唤醒的进程从所在的等待队列中移出，如果没有指定进程，则从等待队列中按照一定策略选择一个进程移出。

### 2. 修改 PCB 的进程状态

把移出进程的 PCB 的状态改为就绪状态。

### 3. 将进程加入合适的就绪队列

根据进程的性质，将所唤醒的进程加入一个合适的就绪队列中。

在 3.3.2 节中曾经提出一个问题：在图 3-1 中，进程的阻塞状态转换为就绪状态时，能否把进程的阻塞状态转换为运行状态而让进程直接运行呢？

如果在一个进程被唤醒时，将它的阻塞状态直接改为运行状态，那么，唤醒原语的操作就要修改如下。

(1) 从等待队列中移出进程

需要唤醒的进程从所在的等待队列中移出，如果没有指定进程，则从等待队列中按照一定策略选择一个进程移出。

(2) 修改 PCB 进程状态

把所唤醒的进程的状态设置为运行状态。

(3) 处理器分配

把处理器分配给这个进程。

作为原语的三个操作，我们可以发现，上述操作(1)和(2)在任何情况下都可以完成，但是操作(3)只有在当前处理器空闲时才能成功，否则执行不成功。因为在多个进程并发执行的情况下，处理器当前可能不是空闲的，即处理器正在执行其他的某个进程。如果这样，操作(3)的执行就不能成功。而原语要求这里三个操作要一致，所以，在(3)的执行不成功时，就要恢复已经执行的(1)和(2)操作，即把进程状态再修改回阻塞状态，再加入原来的等待队列中，并期待下一次再尝试执行这个唤醒原语的操作。

经过上述分析得到，修改后的唤醒原语可能存在“测试”现象，这种现象极大地影响了系统的开销，操作系统不能把阻塞状态的进程直接转化为运行状态。

而原来的唤醒原语中的三个操作在任何情况下都可以执行完成。所以，阻塞状态的进程在唤醒后是转化为就绪状态，而不是直接转化为运行状态。

另外，进程控制还包括任务切换。任务切换主要是由硬件实现，处理器在进行任务切换时，还要进行系统安全保护控制，这方面内容请参见 7.2.2 节。

## 3.6 进程同步

程序的并发执行破坏了程序的可再现性，正确的程序执行后得到的结果却可能是错误的，对于用户来说这是不允许的。本节将介绍进程管理的第二个功能——进程同步，讨论造成这个错误的原因及其解决方法。

### 3.6.1 并发进程的关系

人们经过大量的实践和分析后发现，多个并发执行的进程，存在两种类型的关系：无关的和相关的。无关的进程之间在并发执行后，可以保证程序的可再现性，只有相关的进程之间并发执行后，才可能破坏程序的可再现性。

对于相关的并发进程，又分两种相互制约关系，这两种制约关系的进程，在并发执行过程中，某些特殊指令的轮流交替运行可能导致程序运行的错误。下面先看两个例子。

#### 1. 两个例子

**例 3-2** 假定用户 1 和用户 2 各有一道程序 P1、P2，这两道程序在运行过程中需要把处理的结果通过同一台打印机输出。一般地，用户 1 的程序 P1 处理的结果数据必须连续地打印在纸张上，用户 2 的程序 P2 处理的结果数据也必须连续地打印在纸张上。如表 3-1 所示，描述了程序 P1 和 P2 的打印操作（关于具体打印机处理程序，参见 7.2.1 节的例子）。

**表 3-1 例 3-2 两道程序的打印操作**

| P1 程序 | P2 程序 |
|---|---|
| ⋮ | ⋮ |
| 打印第 1 行 A1 | 打印第 1 行 B1 |
| 打印第 2 行 A2 | 打印第 2 行 B2 |
| ⋮ | ⋮ |
| 打印第 $n$ 行 A$n$ | 打印第 $m$ 行 B$m$ |
| … | … |

那么，在顺序执行的工作流程中，它们都可以正常地运行完成并得到正确的结果，如图 3-3 的(a)和(b)所示，描述了 P1 和 P2 运行的结果，这样，用户 1 和用户 2 各自得到一份打印的数据文档。

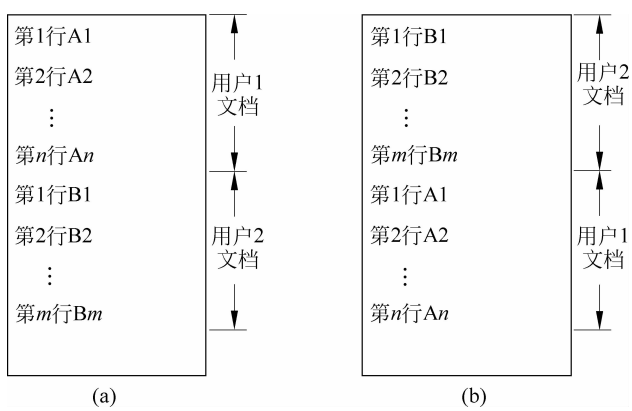



图 3-3 例 3-2 进程顺序执行的结果

然而，在并发执行的工作流程中，P1 和 P2 程序在表面上可以运行完成，但是，运行后打印出来的结果数据可能不是用户所期望的。也就是说，打印纸上用户 1 和用户 2 的数据可

能无法独立地分成两大部分，造成用户 1 的数据与用户 2 的数据混杂在一起，如图 3-4 所示描述了一种可能的打印结果。

这是因为，P1 和 P2 是轮流交替地在处理器上执行的，处理器在执行 P1 和 P2 程序指令的时候，可能存在如下的执行顺序：先执行 P1 中的“打印第 1 行 A1”和“打印第 2 行 A2”操作，之后处理器转向执行 P2，在执行 P2 的“打印第 1 行 B1”后，接着，处理器又转向执行 P1，执行 P1 的“打印第 3 行 A3”，等等。这样，打印纸上用户 1 和用户 2 的数据交织在一起，甚至难以区分哪些是用户 1 的数据、哪些是用户 2 的数据。

第1行 A1
第2行 A2
第1行 B1
第3行 A3
⋮
第$m$行 B$m$
第$n$行 A$n$

图 3-4　例 3-2 进程并发执行的一种结果

接下来，再看另一个例子。

**例 3-3**　假定有一项任务需要将同一台计算机上的一个磁盘的文件备份到另一个磁盘上。

一种实现的方法是：设计一个进程，它每次从源磁盘上读一个文件，然后将所读文件写入目标磁盘，如此反复，直到所有文件备份完成。

这种方法比较简单、容易实现，且可以完成任务的要求。但是，这种方法缺乏并行性，因为进程在执行源磁盘的一个读 I/O 操作时，进程进入阻塞状态，直到数据读入内存后，进程被唤醒转换为就绪状态；然后，当继续运行，执行目标磁盘的写 I/O 操作时，进程再次进入阻塞状态，直到写操作完成，如此反复，实现文件的备份。可见，在这种方式中，两个磁盘的读操作和写操作不能同时进行，也就是缺乏并行性。

实现的另一种方法是：把这个备份的任务分解为三个子任务，分别对应三个进程 Read、Move、Write。进程 Read 每次从源磁盘上读一个文件存入缓冲区 buf1 中，进程 Move 每次把 buf1 中的数据转移到 buf2，进程 Write 将 buf2 中的数据写入目标文件。通过这三个进程的反复执行实现备份的任务，如图 3-5 所示。

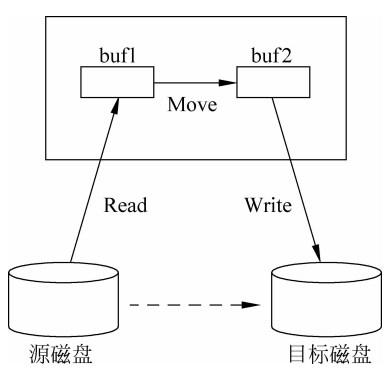



图 3-5　三个进程的任务协作

进程 Read 读一个文件至 buf1（这里不妨假定缓冲区 buf1 或 buf2 能够存储一个文件数据），进程 Move 把 buf1 的文件数据移至 buf2，进程 Write 将 buf2 中的文件数据写入目标磁盘。在进程 Write 执行写操作的同时，进程 Read 可以读下一个文件。一般地，第 $i$ 个文件的 $\text{Move}_i$ 完成后，$\text{Write}_i$ 可以开始，与此同时第 $i+1$ 个文件的 $\text{Read}_{i+1}$ 也可以开始，从而实现了源磁盘的读操作和目标磁盘的写操作同时进行，也就是实现了设备与设备的并行。如

图 3-6 所示，这样每个文件依次经过 Read→Move→Write 后完成了备份操作。

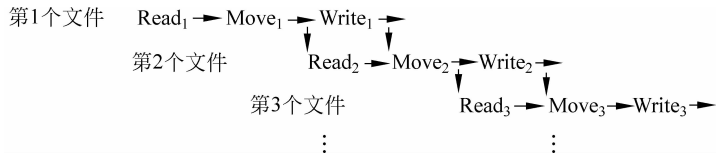



图 3-6 Read、Move、Write 的并行执行

在并发执行方式中，三个进程 Read、Move、Write，执行顺序是随机的，处理器不一定会按照任务期望的顺序 Read→Move→Write 依次执行。

开始时，如果进程 Move 或 Write 先于进程 Read 而运行，则 buf1 和 buf2 中的数据都不是进程期望的数据，这样执行的结果，造成目标磁盘产生多余数据。

另外，在进程 Read 读一个文件至 buf1 后，它可能很快地再次运行，且可能是在进程 Move 把缓冲区 buf1 中文件数据读出之前，进程 Read 新读的一个文件也存入了 buf1，这样执行的结果，造成原来的文件没有写入目标磁盘，导致文件丢失。

还有其他的轮流交替的运行方式，也会造成备份操作的结果错误。例如进程 Move 或 Write 很快地连续运行两次等，这里就不一一分析了。

例 3-2 中的错误结果与 $P_1$ 和 $P_2$ 进程共享同一台打印机有关。当 $P_1$ 中的打印操作与进程 $P_2$ 的打印操作在处理器上轮流交替执行时，就出现了打印结果混乱的情况。

如果在 $P_1$ 和 $P_2$ 的并发执行过程中，能够对处理器的轮流运行进行合理的控制，就可以保证得到正确的结果。例如，处理器开始执行 $P_1$ 的第 1 个打印操作后，如果在 $P_1$ 的所有打印操作完成之前，处理器转向 $P_2$ 执行，那么，不允许处理器执行 $P_2$ 的打印操作，但允许执行 $P_2$ 的其他非打印操作，这样就可以保证打印后数据的独立性，也就得到正确的打印结果。宏观上，进程 $P_1$ 和 $P_2$ 中，当一个进程在使用打印机时，另一个进程就不能使用打印机。

例 3-3 中三个进程的任务协作，如果处理器能够按图 3-6 所示的期望顺序执行，就可以正确完成备份的工作。由于并发执行的随机性，处理器不一定都能按期望的顺序执行，从而造成错误的备份结果。

### 2. 并发进程的制约关系

通过上述的分析，把相关的并发进程的两种相互制约关系称为间接制约关系和直接制约关系。对于这两种制约关系的并发进程，操作系统需要进行合理的控制。

(1) 间接制约关系

两个或多个进程共享一种资源时，当一个进程在访问或使用该资源时，制约其他进程的访问或使用，否则，就可能造成执行结果的错误。把并发进程之间的这种制约关系称为间接制约关系，也就是一个进程通过第三方即共享的资源，暂时限制其他进程的运行。

间接制约关系是由资源共享引起的。

(2) 直接制约关系

直接制约关系则是由任务协作引起的。几个进程共同协作完成一项任务，因任务性质的要求，这些进程的执行顺序有严格的规定，只有按事先规定的顺序依次执行，任务才能得

到正确的处理，否则，就可能造成错误结果。把并发进程之间的这种制约关系称为直接制约关系，也就是一个进程的执行状况直接决定了另一个或几个进程可否执行。

一组进程（以后没有特殊指明，一组进程就是指两个或两个以上的进程）如果存在间接制约或直接制约关系，那么它们在并发执行时，微观上的轮流交替就要受到限制，需要操作系统合理地控制它们的工作流程，以保证执行结果的正确性。

为了实现对并发执行的控制，操作系统需要从程序代码上分析进程的制约关系，提出对轮流交替实施控制的方法。

## 3.6.2　间接制约与互斥关系

并发进程之间的间接制约关系是一种最单、最基本的制约关系。

### 1. 资源的使用步骤

在操作系统的资源管理下，用户程序使用资源的步骤是：申请、使用、归还，如图 3-7 所示。

用户程序使用资源时，首先向操作系统提出申请，操作系统根据当前系统的状况进行资源分配，在得到资源后，用户才能使用资源，在一次申请得到资源后，用户可以根据需要分多次使用。最后，当用户不再使用资源时，用户程序必须归还已申请得到的资源。

这里需要说明的是，上述提到的申请、使用和归还的操作都是通过操作系统提供的系统调用实现的。对于不同资源，这些操作所对应的系统调用有所差别。并且，程序员在使用高级语言编写程序时，对于一些特殊资源的使用，为了简化程序员的编程过程，程序中可以没有申请或归还的对应语句，由编译系统在编译过程中自动补充申请或归还资源的操作。

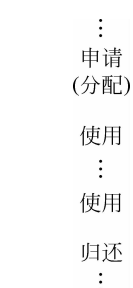



图 3-7　资源使用步骤

### 2. 临界资源与间接制约

一次只能让一个进程使用的资源称为临界资源，这里“一次”的含义，需要从资源使用步骤的角度来理解。在一个进程申请、分配得到资源起，到归还资源为止的时间段内，进程对该资源的使用过程称为一次使用。

常见的临界资源有打印机、存储单元、堆栈、链表、文件等。

间接制约关系就是一组并发进程在共享某种临界资源时存在的一种制约关系，例 3-2 中的 $P_1$ 和 $P_2$ 就具有间接制约关系。

### 3. 临界区与互斥关系

为了实现对间接制约关系的控制，需要从代码上进一步分析这种关系，为此引入临界区的概念。

临界区（Critical Section，或 Critical Region）是指进程对应的程序中访问临界资源的一段程序代码，就是进程在资源的一次使用过程中，从申请开始至归还为止的一段程序代码。

在需要多个临界资源的情况下，多个进程之间的临界区还分为相关临界区和无关临界区。两个或多个临界区称为相关临界区，指这些临界区访问的是同一个临界资源。无关临界区是不同临界资源之间的临界区。

以后如果没有特殊说明，临界区是指相关临界区。

引入临界区的目的是对并发进程的间接制约关系进行控制。

对于一个临界区，称一个进程要进入临界区执行，是指该进程即将要运行临界区的第一条指令/语句；称一个进程离开或退出临界区，是指该进程已经执行了临界区的最后一条指令/语句；称一个进程在临界区内执行，是指该进程已经开始执行临界区的第一条指令但还没有离开这个临界区。

对于一组并发进程的临界区，进程之间对临界区的执行需要互斥执行，即至多只能有一个进程在临界区内执行，当有一个进程在临界区内执行时，其他要进入临界区执行的进程必须等待。也就是说，不允许处理器在临界区之间轮流交替地执行。

两个或两个以上的一组并发进程，称它们具有互斥关系，是指这组进程至少共享一类临界资源，当一个进程在临界资源对应的临界区内执行时，其他要求进入相关临界区执行的进程必须等待。

操作系统对一组进程的间接制约关系的控制，转为实现这组进程的互斥关系。具有互斥关系的一组进程也称为互斥进程。

### 3.6.3 直接制约与同步关系

并发进程之间的直接制约关系是一种常见的制约关系。由于多个进程的任务协作产生了执行顺序上的依赖关系，这种顺序不一定是整个进程间的依赖关系，也可以是进程内部某些指令间的执行先后顺序的规定。因此，多道程序设计的并发执行方式，不仅可以提高系统的资源利用率，而且为多个进程的任务协作提供了可能。

#### 1. 单向依赖关系

对于进程 A 和 B，如果处理器在执行进程 A 中某条指令之前，要求先执行进程 B 的一条指令；在进程 B 指定的指令没有执行之前，进程 A 的对应指令不能执行，这时称进程 A 依赖于进程 B。如图 3-8 所示，进程 A 的 L1 这条指令的执行依赖于进程 B 的 L2 指令的执行，进程 A 依赖于进程 B，但进程 B 的执行不受限制。

#### 2. 相互依赖关系

如果进程 A 依赖于进程 B，同时进程 B 也依赖于进程 A，则称进程 A 和 B 具有相互依赖关系。如图 3-9 所示，进程 A 中 L1 指令的执行依赖于进程 B 的 L4 指令，而进程 B 的 L3 指令的执行也依赖于进程 A 的 L2 指令。

相互依赖关系主要是由于任务的反复执行而产生的。例如，在初始状态下，进程 A 的第一次执行不受限；但是，如果进程 A 很快地再次执行，就要依赖于进程 B 的执行，而进程 B 的第一次执行就要依赖于进程 A。

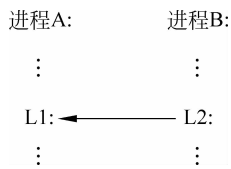



图 3-8　单向依赖关系

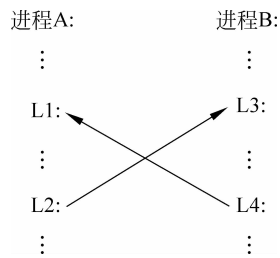



图 3-9　相互依赖关系

### 3. 同步关系

在一组并发进程中，如果每个进程至少与同组中另一个进程存在单向或相互依赖关系，则称这组进程具有同步关系，简称同步进程。

## 3.6.4　进程同步机制

对于具有互斥或同步关系的进程，操作系统要采用措施对它们的轮流交替方式进行控制，以保证各进程执行结果的正确。把用于控制并发进程的互斥、同步关系，保证它们能够正确执行的方法称为进程同步机制。

常用的进程同步机制有加锁机制、标志位机制、信号量机制和管程机制。

## 3.6.5　互斥关系与加锁机制

操作系统对进程互斥关系的实现，转化为对临界区执行的控制。

### 1. 临界区管理准则

如何验证同步机制在实现进程互斥关系控制时的有效性？人们经过总结，得到临界区管理的 4 个准则。

(1) 空闲让进

在一个进程要求进入临界区执行时，如果没有进程在相关临界区内执行，则应允许其进入临界区运行，这是提高资源利用率的体现。

所谓进程要求进入临界区执行，是指该进程接下来就要执行临界区的第一条指令。

(2) 忙则等待

当有一个进程在临界区内执行时，要求进入相关临界区执行的其他任何进程都要等待，这是互斥关系的体现。

进程在临界区内执行，是指进程已经开始执行临界区的第一条指令，但临界区的最后一条指令还没有执行完成。

(3) 有限等待

对于要求进入临界区执行的进程，至多经过有限时间的等待之后，应有机会进入临界区执行，不能让其无期限地等待下去。“有限等待”是为了系统的公平性。

（4）让权等待

当进程离开(或者退出)临界区时，即在临界区内执行的进程在执行完临界区的最后一条指令后，应把处理器让给下一个进程执行。“让权等待”与进程调度密切相关。

临界区管理的准则也为实现互斥提供了思路，即每个进程在要求进入临界区执行时，同步机制必须进行检查，只有在得到同步机制的许可后，才能进入临界区执行，否则必须等待；在进程离开临界区后，同步机制也需要检查，是否有其他进程在等待中，如果发现有进程在等待，则可以选择一个让它进入临界区执行。

### 2. 加锁机制原理

人们借鉴日常生活中所熟悉的锁对门的控制作用，而得到加锁机制，其原理的主要内容如下。

（1）锁变量 key

对于一组相关临界区定义一个变量称为锁变量，锁变量取值 0 或 1，并规定 key＝0 时表示对应的锁是开的，临界资源当前是空闲的，此时允许进程进入对应的临界区执行；key＝1 表示对应的锁是关的，临界资源当前是忙的，此时禁止进程进入对应的临界区。

（2）加锁操作 lock(key)

加锁操作定义如下：

```
lock(key)
{
    while(key == 1);
    key = 1;
}
```

为叙述方便，以后把这里“while(key＝＝1);”称为循环测试语句；“key＝1;”称为设置语句，加锁操作也就是由这两条语句组成的，即循环测试语句和设置语句。

加锁操作的作用是检查进程是否可以进入临界区执行。在临界区的第一条指令之前，加入一个加锁操作，以实现进程要进入临界区执行时的检查。

一个进程在执行 lock(key)操作时，如果 lock()运行完成，则称为加锁成功，意味着该进程得到锁，只有得到锁的进程才允许进入临界区执行，没有得到锁的进程要等待。

一个得到锁的进程离开临界区时，利用解锁操作，归还锁变量。

（3）解锁操作 unlock(key)

解锁操作的定义如下：

```
unlock(key)
{
    key = 0;
}
```

### 3. 加锁机制应用及例子

假定 $p_1$、$p_2$、…、$p_n$是一组互斥关系的进程，对应的锁变量为 key，那么，加锁机制的应用方法如下。

置锁变量初值 key＝0，对于进程 $p_i$，$i=1,2,\cdots,n$，其加锁机制的控制方法描述如下：

```
…
lock(key);
临界区;
unlock(key);
…
```

为了分析锁机制的有效性，下面，应用加锁机制对 3.1.3 节的例 3-1 中进程 PA()和 PB()进行控制。

```
PA()
{
      int x;
      lock(key);
  ①   x = count;
  ②   x = x + 1;
  ③   count = x;
      unlock(key);
}
PB()
{
     int y;
     lock(key);
  ④   y = count;
  ⑤   y = y - 1;
  ⑥   count = y;
      unlock(key);
}
```

先分析 3.1.3 节中提出的按①④⑤⑥②③执行的情况是否能够被控制。

处理器在执行 PA()的①之前，应先执行 PA()中的 lock(key)加锁操作，由于这时 key＝0，所以，PA()执行 lock(key)中的循环测试语句时，因循环条件不成立，循环测试语句运行结束，接着执行设置语句，并返回。此时，PA()得到锁，置 key＝1，进入临界区，处理器执行①。现在按前面的假定，处理器要转去执行 PB()的④⑤⑥，从上述代码可知，在执行④之前，应先执行 PB()的 lock(key)加锁操作，当处理器在执行 PB()中 lock(key)的循环测试语句时，因为 key＝1，所以循环条件一直成立，处理器就不断地执行这条循环测试语句，因而暂时无法执行 PB()的④及其后续的语句。这样，只有在将来处理器轮到 PA()，继续执行 PA()的②、③和 unlock(key)后，key＝0，PA()离开临界区，之后当处理器再次轮到 PB()执行时，PB()才能进入临界区执行。

从上述的分析来看，3.1.3 节中提出的按①④⑤⑥②③执行的情况似乎不会出现。

那么，加锁机制可以实现互斥关系吗？

人们进一步分析发现，加锁操作 lock(key)中的两条语句的执行可能被分割，也就是说，当处理器在执行 PA()的 lock(key)中的循环测试语句后，还没有来得及执行设置语句，处理器就转而执行 PB()的 lock(key)操作，因之前 PA()的 lock(key)操作还没有设置 key 的值，所以，仍有 key＝0，这样 PB()的 lock(key)操作立即执行完成，PB()得到锁，而进入临界

区。之后，如果在 PB()没有离开临界区之前，处理器转向执行 PA()，PA()的现场恢复后，执行 lock(key)中的设置语句并返回，PA()也得到锁，同时进入临界区。

也就是说，在上述的加锁机制控制下，3.1.3 节中提出的按①④⑤⑥②③执行的情况还是不能得到控制。

所以，加锁机制不能满足临界区管理的准则(2)，也就是不能实现互斥关系。

但是，如果加锁操作借助硬件实现，就可以实现进程的互斥关系。

这里以 x86 为例，利用汇编指令 xchg 实现 lock(key)，8086 汇编语言描述如下：

```
tsl:
      mov ax,1
      xchg ax,key
      cmp ax,0
      jne tsl
```

对于多处理器系统，通常提供指令前缀 lock，利用指令前缀 lock 封锁总线实现指令的执行的互斥，具体描述如下：

```
tsl:
      mov ax,1
lock xchg ax,key
      cmp ax,0
      jne tsl
```

可以验证，经过这样修改后，加锁机制可以实现互斥关系。

#### 4. 加锁机制分析

通过对上述例子的分析，得到以下结论。

(1) 普通的加锁机制不能实现互斥关系，借助硬件的加锁机制可以实现进程的互斥关系。

(2) 存在“忙等待”现象，浪费了处理器时间。

当 key=1 时，处理器执行一个进程的 lock(key)操作的结果是反复地执行循环测试语句。处理器虽然分配给这个进程，但只是循环地执行这个测试操作，而且只能期待其他进程离开或退出临界区时，执行 unlock(key)操作归还锁变量之后，循环测试才能结束，而这时其他进程又暂时没有机会得到处理器运行。所以，浪费了处理器的时间。这种现象称为进程的“忙等待”(Busy Waiting)。

(3) 存在“饥饿”现象。

一个进程在“忙等待”时，它期待将来其他某一个进程的 unlock(key)操作，这种期待可能会无期限地等待，原因是，虽然将来轮到其他的一个进程执行时，执行了 unlock(key)操作置锁变量 key=0，但是，处理器可能轮到其他的第三个进程，第三个进程正好也要执行加锁操作，并成功得到锁，进入临界区执行，在第三个进程退出临界区之前，处理器轮到原来“忙等待”进程执行，这时又有 key=1，所以，它仍然只能“忙等待”。这样的状况可能反复地出现，造成这个“忙等待”的进程一直在加锁操作中等待。这种状况被称为“饥饿”(Starvation)现象，或“饿死”现象，可见，加锁机制不满足临界区管理准则(3)。

(4) 多个锁变量的加锁操作可能造成进程死锁。

死锁是并发执行方式存在的另一种形式的错误,多个锁变量的加锁操作可能造成进程死锁。死锁问题将在 4.4 节中专门介绍。

### 3.6.6 信号量机制与互斥关系

信号量机制是由荷兰计算机专家 Dijkstra 在 1965 年提出的,他借鉴交通路口的信号灯控制来往车辆的作用,设计了信号量机制。信号量机制的内容虽然简单,但应用广泛,不仅可以实现进程互斥关系,还可以实现进程同步关系。

#### 1. 信号量机制原理

信号量机制原理如下。

(1) 信号量

信号量是一种变量,一个信号量对应一个整型变量 value、一个等待队列 bq,同时还可以对应其他的控制信息。为了简便起见,这里把信号量的数据类型简化定义如下。

```
struct semaphore {
    int value;
    PCB * bq;
}
```

其中,value 是信号量对应的整型变量,bq 是信号量对应的等待队列。

信号量数据类型是操作系统的关键数据结构之一,组织在内核中。

(2) p 操作

s 是一个信号量,p 操作定义如下。

```
p(s)
{
 s.value = s.value - 1;
 if (s.value < 0) blocked(s);
}
```

这里 blocked(s)是阻塞原语,把当前调用 p(s)操作的进程设置为阻塞状态并加入到信号量 s 对应的等待队列 bq 中。

(3) v 操作

s 是一个信号量,v 操作定义如下。

```
v(s)
{
  s.value = s.value + 1;
  if (s.value <= 0) wakeup(s);
}
```

这里 wakeup(s)是唤醒原语,从信号量 s 对应的等待队列 bq 中唤醒一个进程,也就是按一定策略从等待队列 bq 中选择一个进程,将其转化就绪状态。

在信号量机制中,p 操作和 v 操作定义为原语。

### 2. 信号量机制分析

假定 s 是一个信号量，从 p、v 操作的定义可知：当 s.value≥1 时，进程调用 p(s)操作后，不会造成进程阻塞；当 s.value≤0 时，进程调用 p(s)操作后，将造成进程阻塞。所以，p 操作具有限制的作用。

另外，当 s.value≤0 时，进程调用 p(s)操作的结果是进程进入阻塞状态，系统把处理器分配给下一个进程运行，而不是像加锁机制中的“忙等待”。

因调用 p(s)操作而进入阻塞状态的进程，在合适的时候，可以通过其他进程的 v(s)操作被唤醒。因为，如果进程在执行 p(s)操作后，进入阻塞状态，进程被加入到 s.bq 的等待队列中，这时有 s.value<0。之后，如果处理器执行了其他进程的一个 v(s)操作，根据 v 操作的定义，此时必然有 s.value≤0 成立，说明 v(s)操作内部一定会调用 wakeup(s)的操作，从 s.bq 等待队列中唤醒一个阻塞状态的进程。只要能够合理地应用 p 操作和 v 操作，由 p 操作阻塞的进程就都会由 v 操作唤醒。

信号量机制可以避免加锁机制中的“饥饿”现象，因为当 v(s)操作需要从 s.bq 等待队列中唤醒一个进程时，可以采取一些策略(如先进先出等)，保证 s.bq 中的每个进程都有机会被唤醒，避免某个进程无期限地等待下去。

### 3. 信号量机制实现互斥关系

假定进程 $p_1$、$p_2$、…、$p_n$共享某一个临界资源，定义一个信号量 s，初值为 1，那么，应用信号量机制实现 $p_1$、$p_2$、…、$p_n$互斥关系的模型如下。

对于进程 $p_i$，$i=1,\cdots,n$，其信号量机制的控制描述如下。

```
…
p(s);
临界区;
v(s);
…
```

也就是说，一组相关的临界区定义一个信号量，在每个进程临界区的第一条指令之前加入一个 p 操作，在临界区的最后一条指令之后加入一个 v 操作。通常，把用于描述、控制临界区互斥的信号量称为互斥信号量。

下面应用这个模型，也来实现对 3.1.3 节的例 3-1 中 PA()和 PB()进程的并发控制。

定义信号量：semaphore s=1；

```
PA()
    {
            int x;
            p(s);
        ①   x = count;
        ②   x = x + 1;
        ③   count = x;
            v(s);
}
PB()
```

```
{
        int y;
        p(s);
    ④   y = count;
    ⑤   y = y - 1;
    ⑥   count = y;
        v(s);
}
```

在上述信号量机制的控制下，再来分析 3.1.3 节中提出的按①④⑤⑥②③执行的情况是否能够被控制。

处理器在执行 PA()的①之前，应先执行 PA()中的 p(s)操作，由于这时 s.value＝1，所以执行 p(s)后，s.value＝0，不会引起进程阻塞，p(s)操作很快返回，PA()进入临界区，处理器执行①。现在，按照假定，处理器转去执行 PB()的④⑤⑥，从上述代码可知，处理器在执行④之前，应先执行 PB()的 p(s)操作，当处理器执行 PB()中 p(s)之前，s.value＝0，所以，在 p(s)执行后 s.value＝－1，PB()进程因 p(s)操作进入阻塞状态，PB()被加入 s.bq 的等待队列中，而暂时无法执行 PB()的④及其后续的语句。这样，将来处理器轮到 PA()，继续执行 PA()的②、③和 v(s)，并离开了临界区。处理器在执行 PA()的 v(s)前，s.value＝－1，所以，PA()的 v(s)操作的执行将从 s.bq 等待队列中唤醒阻塞状态的进程 PB()。当处理器再次轮到 PB()执行时，执行 PB()中 p(s)的下一条指令，即 PB()进入临界区执行。

由此可见，3.1.3 节中提出的按①④⑤⑥②③执行的情况不会出现。同样，3.1.3 节中提出的按④①②③⑤⑥执行的情况也不会出现。

这样，那些可能导致错误结果的轮流交替都被控制了。另外，因为 p 操作和 v 操作是原语，所以不会出现如加锁操作的两条语句被分割执行的情况，所以，实现了 PA()和 PB()的互斥执行。

同样，3.6.1 节中的例 3-2 中的 P1 和 P2 因共享打印机，应用信号量机制可以很方便地实现它们的互斥执行，如表 3-2 所示。

**表 3-2　3.6.1 节中例 3-2 的并发程序设计**

| **semaphore s＝1；** | |
|---|---|
| P1 程序 | P2 程序 |
| … | … |
| p(s)； | p(s)； |
| 打印第 1 行 A1 | 打印第 1 行 B1 |
| 打印第 2 行 A2 | 打印第 2 行 B2 |
| … | … |
| 打印第 *n* 行 A*n* | 打印第 *m* 行 B*m* |
| v(s)； | v(s)； |
| … | … |

## 3.6.7　信号量机制与同步关系

信号量机制不仅可以实现进程的互斥关系，还可以实现进程的同步关系。这里，就

3.6.3 节中的两种依赖关系，用信号量机制进行控制。

### 1. 简单同步关系

单向依赖关系也称为简单同步关系。对于如图 3-8 所示的进程 A 和 B，进程 A 的 L1 的执行依赖进程 B 的 L2 的执行。定义一个信号量 s 对应这个依赖关系，s 的初值为 0，信号量机制的控制描述如图 3-10 所示。

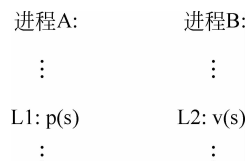


图 3-10 简单同步关系

下面对图 3-10 做简要分析。

如果处理器先执行进程 A，当执行 L1 的 p(s)时，因为 s.value＝0，所以 p(s)执行后，s.value＝－1，进程 A 进入阻塞状态，这是因为，此时进程 B 的 L2 还没有执行。进程 A 的阻塞状态何时被唤醒？在将来处理器轮到进程 B 执行，在执行 L2 的v(s)时，因为这时 s.value＝－1，所以，v(s)操作的执行唤醒了在 s.bq 等待的进程 A，这样，在将来进程 A 继续运行时，进程 B 已经执行了 L2。所以，处理器执行 A 和 B 的顺序，符合进程 A 和 B 的依赖关系。

如果处理器先执行进程 B，当执行了 L2 的 v(s)时，s.value＝0，所以 v(s)执行后，s.value＝1，这个 v(s)操作不会执行唤醒原语，因为这时 s.value＝1，且进程 A 还没有进入阻塞状态。之后，处理器如果转向执行进程 A，在执行 L1 时，因为 s.value＝1，所以，在 p(s)执行后，s.value＝0，进程 A 不会被 p(s)操作阻塞而继续运行。同样，处理器执行 A 和 B 的顺序，也符合进程 A 和 B 的依赖关系。

因此，在如图 3-10 所示的控制方式下，在进程 A 和 B 轮流交替运行时，处理器都能按事先期望的顺序执行。

### 2. 一般同步关系

相互依赖关系也称为一般同步关系。对于如图 3-9 所示的进程 A 和 B，应用上述简单同步关系的方法，容易得到一般同步关系的信号量机制的实现。

一个依赖关系定义一个信号量，这里有两个依赖关系，所以，需要定义两个信号量。定义 semaphore s1＝1，s2＝0（假定进程 A 的第一次执行不受限制），进程 A 和进程 B 的控制描述如图 3-11 所示。

通常，把用于描述、控制依赖关系的信号量称为同步信号量。

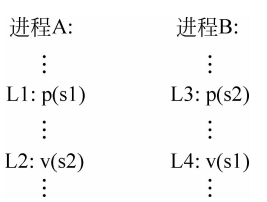


图 3-11 一般同步关系

### 3. 并发程序设计

应用同步机制描述对进程的并发控制称为并发程序设计（Concurrent Programming）。

基于信号量机制的并发程序设计就是利用信号量及 p 操作和 v 操作描述对进程的并发控制。并发程序设计侧重于对进程之间的轮流交替的控制，使得处理器在执行这些进程时能够以正确结果的方式执行，避免那些可能造成错误的轮流交替执行方式，在并发程序设计中，可以不必过于关心进程功能的实现细节。

在并发程序设计中，用 cobegin{ }描述并发执行的一组进程。

这里举两个例子，说明并发程序的设计方法。之后在 3.6.8 节和 3.6.9 节两节中再介绍两个经典同步问题。

**例 3-4** 两个进程 P1 和 P2 共享一个缓冲区 buf，进程 P1 反复地计算，并把计算结果存入缓冲区 buf，进程 P2 每次从缓冲区中取出计算结果并送向打印机。规定：P1 把结果存入缓冲区 buf 后，P2 才能打印，P1 的一次计算的结果只能打印一次，只有在结果被打印后，P1 新的计算结果才能存入缓冲区。试用信号量机制实现 P1 和 P2 的并发执行。

分析：进程 P1 和 P2 的关系为一般同步关系。因为进程 P1 中“计算结果存入 buf”的操作依赖于进程 P2“从 buf 取出结果”操作，但 P1 的第一次执行可以不受限制，因为可以假定开始时缓冲区 buf 是空的；同时，P2 的“从 buf 取出结果”的操作又依赖于 P1 的“计算结果存入 buf”的操作。

**解**：并发程序设计如下。

```
semaphore s1 = 1,s2 = 0;
P1(){
    计算并得到结果;
    p(s1);
    结果存入缓冲区 buf;
    v(s2);
    }
P2(){
    p(s2);
    从缓冲区 buf 取出结果;
    v(s1);
    打印结果;
    }
main(){
    cobegin {
        repeat P1();
        repeat P2();
        }
    }
```

这里，需要特别强调指出以下几点。

(1) 并发程序设计可以不需要过分关注实现细节。

例如，P1 计算的结果是什么类型的数据、数据量有多少、缓冲区 buf 可否足够存储所计算的结果，还有，如打印格式怎样等，这些实现细节在并发程序设计中可以不考虑。

(2) 进程 P1 和 P2 如何能对同一个缓冲区 buf 进行存、取操作。

在 3.2.2 节中提到进程具有独立性，那么，进程 P1 和 P2 又如何能对同一个缓冲区 buf 进行存、取操作？

可以从两个方面来解释，一方面，因为本书讲解操作系统的原理和管理、控制方法，可以把例 3-4 中的进程看成是操作系统内核的代码，它们运行在核心态下，所以可以访问所需要的资源，包括例 3-4 中的缓冲区 buf。

另一方面，也可以这样理解：进程 P1 和 P2 是用户进程，运行在用户态，但是，其中的“结果存入缓冲区 buf”和“从缓冲区 buf 取出结果”操作视为操作系统的系统调用，这样，上

述并发程序设计中的 p 操作和 v 操作看成是在系统调用内部，如“结果存入缓冲区 buf”的系统调用内部描述为：

```
p(s1);
结果存入缓冲区 buf;
v(s2);
```

而“从缓冲区 buf 取出结果”的系统调用内部描述为：

```
p(s2);
从缓冲区 buf 取出结果;
v(s1);
```

因此，并发程序设计的重点是描述并发执行时轮流交替的控制方法，可以简化进程具体功能的实现细节。

**例 3-5** 试用信号量机制实现 3.6.1 节中例 3-3 中的第二种方法，实现 Read、Move、Write 的并发执行。

**解：**

```
semaphore s1 = 1,s2 = 0,s3 = 1,s4 = 0;
Read()
{
    从源磁盘上读一个文件;
    p(s1);
    文件数据存入缓冲区 buf1;
    v(s2);
}
Move()
{
    p(s2);
    从缓冲区 buf1 取文件数据;
    v(s1);
    p(s3);
    将文件数据存入 buf2;
    v(s4);
}
Write()
{
    p(s4);
    把 buf2 中的数据存入目标磁盘的文件中;
    v(s3);
}
main(){
    cobegin
    {
        repeat Read();
        repeat Move();
        repeat Write();
    }
}
```

### 3.6.8 生产者/消费者问题

生产者/消费者问题(简称 PC 问题)是最经典的同步问题,很多同步问题经过抽象后都可以转化为生产者/消费者问题。本节作为并发程序设计的例子,将介绍 PC 问题及其并发程序设计描述。

#### 1. PC 问题

人类进入工业社会以来,物质资料的生产、流通和消费成为社会活动的主体。在不考虑流通环节的情况下,就简化为生产和消费方式,生产者不断地生产物品,生产的物品存入仓库的货位上,消费者从仓库的货位上取出物品进行消费。虽然生产者和消费者的工作都有很大的随机性,但是借助于仓库及其货位,生产者和消费者自然地就可以协调地工作,不会产生什么问题。例如,在仓库存满、没有空货位的情况下,生产者无法把新物品放入仓库,这时,允许生产者在仓库门口外等待,等待消费者取出物品,空出货位后再存入;另外,如果仓库没有物品,消费者没有物品可以取,也可以在门口外等待,等待生产者把物品存入后再取,如果有物品,消费者从仓库一个货位上取出物品,取出后仓库中的物品自然就少一个。然而,把这种生产和消费的工作方式在计算机系统中实现时,就会产生问题。

在计算机系统中,生产者(Producer)、消费者(Consumer)是用进程来模拟的,称为生产者进程、消费者进程,仓库由缓冲区实现,货位就是缓冲区的单元格/区域,物品转化为数据。生产者进程和消费者进程的任务描述如下。

```
Producer()
    {
        生产一个物品;
        物品存入缓冲区;
    }
Consumer()
    {
        从缓冲区取出物品;
        消费;
    }
main()
    {
    cobegin
        {
            repeat Producer();
            repeat Consumer();
        }
    }
```

可以看出,Producer 的“物品存入缓冲区”操作依赖于 Consumer 的“从缓冲区取出物品”的操作,而 Consumer 的“从缓冲区取出物品”操作又依赖于 Producer 的“物品存入缓冲区”的操作。进程 Producer 和 Consumer 的关系为一般同步关系。

那么,在计算机系统中,生产者 Producer 和消费者 Consumer 进程并发执行时,如果没有进行控制,可能存在“物品丢失”和“重复消费”的错误。具体分析如下。

由于 Producer 和 Consumer 并发执行的随机性，可能存在如下两种典型的运行方式。

一种方式是，在消费者进程 Consumer 还没有运行的情况下，生产者进程 Producer 连续运行了多次，导致缓冲区满（即缓冲区各单元格都已存入了物品）的状态，这时，如果生产者 Producer 轻易地把新物品存入缓冲区，就会造成原来物品丢失的错误。

另一种可能的方式是，在初始状态下，消费者进程 Consumer 先于生产者进程 Producer 执行，Consumer 从空的缓冲区中取物品，造成物品“无中生有”的错误。另外，对于缓冲区单元格，进程在执行一次写操作后，可以执行多次的读操作，所以，当消费者进程 Consumer 从缓冲区的一个单元格上取出物品数据后，在生产者进程 Producer 存入新物品前，消费者进程 Consumer 如果再次从这个单元格取物品数据，这就造成一个物品的两次或多次消费，即重复消费的错误。

所以，操作系统需要控制生产者进程 Producer 和消费者进程 Consumer 并发执行，以保证生产者、消费者正确地协调工作。

### 2. PC 问题分类

假定生产者进程个数为 $n$，消费者进程个数为 $m$，缓冲区单元格个数为 $k$。把 PC 问题分为以下 4 类。

（1）简单 PC 问题

把 $n=1$、$m=1$ 且 $k=1$ 时的 PC 问题称为简单 PC 问题。3.6.7 节中的例 3-4 就是一个简单 PC 问题，其中 P1 相当于生产者进程，P2 相当于消费者进程，计算的结果就是物品，打印操作相当于消费。

（2）一般 PC 问题

把 $n=1$、$m=1$ 且 $k>1$ 时的 PC 问题称为一般 PC 问题，后面将重点介绍。

（3）复杂 PC 问题

把 $n>1$、$m>1$ 且 $k>1$ 时的 PC 问题称为复杂 PC 问题，后面也将重点介绍。

（4）特殊 PC 问题

特别地，把 $k=1$、$n+m=3$ 或 $n+m=4$ 时的 PC 问题称为特殊 PC 问题。一些典型的特殊 PC 问题作为习题，供读者解答。

### 3. 一般 PC 问题的并发程序设计

**例 3-6** 一般 PC 问题：一个生产者进程 Producer 和一个消费者进程 Consumer 共享一个单元格数量为 $k$ 的缓冲区 buf[$k$]，其中 $k>1$。

一般 PC 问题有两种基本的并发设计方法。

解法 1：假设生产者进程 Producer 和消费者进程 Consumer 不能同时访问缓冲区 buf，这时，进程 Producer 和 Consumer 不仅具有一般同步关系，还具有互斥关系。生产者和消费者进程的并发程序设计如下：

```
semaphore mutex = 1, empty = k, full = 0;
Producer()
{
    生产一个物品;
```

```
    p(empty);
    p(mutex);
    物品存入缓冲区 buf[]的某个单元格;
    v(mutex);
    v(full);
}
Consumer()
{
    p(full);
    p(mutex);
    从缓冲区 buf[]的某个单元格取物品;
    v(mutex);
    v(empty);
    消费;
}
```

分析如下。

(1) 信号量初值

信号量定义后,如何设置初值? 信号量可以表示资源的数量,初始状态时,缓冲区的 $k$ 个单元格都没有存放物品,所以同步信号量 full=0,同时也意味着允许 Producer 连续地执行 $k$ 次,第 $k+1$ 次执行时才需要限制,故同步信号量 empty=$k$。

(2) 连续两个或多个 p 操作时的顺序要求

在上述并发程序设计中,有两个连续的 p 操作,这里的两个连续 p 操作顺序至关重要,必须是先执行同步信号量的 p 操作,再执行互斥信号量的 p 操作。否则,将产生并发执行的另一种错误即死锁(第 4 章 4.4 节中将介绍这个问题)。

例如,把生产者进程 Producer 和消费者进程 Consumer 中的两个连续 p 操作都修改如下:

```
Producer(){                    Consumer(){
    …                              …
    p(mutex);                      p(mutex);
    p(empty);                      p(full);
    …                              …
    v(mutex);                      v(mutex);
    v(full);                       v(empty)
}                                  }
```

那么,在初始状态下,如果消费者进程 Consumer 先执行,因为初值 mutex=1,所以消费者进程 Consumer 在执行 p(mutex)后,mutex=0,不会被阻塞,消费者进程 Consumer 继续执行 p(full); 因为初值 full=0,这时消费者进程 Consumer 被阻塞,这符合实际情况,因为此时缓冲区中没有物品。

但是,之后处理器在执行生产者进程 Producer 时,因为 mutex=0,一旦执行它的 p(mutex)操作,就会造成生产者进程 Producer 被阻塞,这就不符合实际情况,因为,这时缓冲区实际上有空的单元格,而生产者进程却因执行 p(mutex)阻塞而不能存放物品。这样,生产者进程 Producer 和消费者进程 Consumer 都进入了阻塞状态,消费者进程 Consumer 等待生产者进程 Producer 把物品存入缓冲区后执行 v(full),而生产者进程 Producer 又等

待消费者进程 Consumer 取出物品后退出临界区执行 v(mutex)，造成生产者进程 Producer 和消费者进程 Consumer 之间互相等待，永远处于阻塞状态。

所以，在并发程序设计中，如果有两个或多个连续的 p 操作，就必须认真分析，合理安排它们的执行顺序，避免出现上述错误。

(3) 缺乏并行性

在解法 1 中，把缓冲区视为临界资源，导致生产者进程 Producer 和消费者进程 Consumer 存在互斥关系。虽然存储单元是临界资源，但在实际工作过程中，生产者进程 Producer 和消费者进程 Consumer 并不会对缓冲区的同一个单元格同时进行操作。如果对每个单元格设置空或满标记，则生产者进程 Producer 只对缓冲区空标记的单元格操作，而消费者进程 Consumer 只对缓冲区满标记的单元格操作。

所以，在一般 PC 问题中，可以不要考虑生产者与消费者的进程互斥关系，而得到解法 2。

解法 2：对每一个缓冲区单元格 buf[$x$]设置空或满的标记。并发程序设计如下：

```
semaphore empty = k,full = 0;
Producer()
{
    生产一个物品;
    p(empty);
    找一个空标记的缓冲区单元格 buf[x];
    物品存入 buf[x];
    设置 buf[x]为满标记;
    v(full);
}
Consumer()
{
    p(full);
    找一个满标记的缓冲区单元格 buf[y];
    从 buf[y]取物品;
    设置 buf[y]为空标记;
    v(empty);
    消费;
}
```

接着，对上述的并发程序设计作简要分析。

(1) 同步关系的理解

在生产者进程 Producer 中“找一个空标记的缓冲区单元格 buf[$x$]”的操作依赖于消费者进程 Consumer 中“设置 buf[$y$]为空标记”，也就是说如果消费者进程 Consumer 的“设置 buf[$y$]为空标记”执行后，生产者进程 Producer 的“找一个空标记的缓冲区单元格 buf[$x$]”的操作就一定能够成功。同步信号量 empty 用于描述、控制这个依赖关系。

又因为缓冲区单元格有 $k$ 个，初始时的标记都是空，允许生产者进程 Producer 一开始连续执行 $k$ 次，所以初值 empty=$k$。

同样，同步信号量 full 用于描述、控制消费者进程 Consumer 中“找一个满标记的缓冲区单元格 buf[$y$]”的操作依赖于 Producer 进程中“设置 buf[$x$]为满标记”的操作，初始时没

有满标记的单元格，所以初值 full＝0。

（2）具有并行性

实现生产者进程 Producer 和消费者进程 Consumer 同时操作。生产者进程 Producer 在向一个单元格存入新物品的同时，消费者进程 Consumer 可以从另一个单元格取出之前由生产者进程存入的物品，从而提高了并行程度。

（3）存、取缓冲区物品的方式

在上述设计中，通过设置标记和查找方式进行存、取物品，如果规定按先进先出方式循环存、取物品，并借助信号量的控制作用，则并发程序设计的描述可以更为简洁。

例如，引入两个变量，分别用于指示当前生产者、消费者可访问的单元格的位置/指针，初值均为 0，每次存或取操作后，对应变量值加 1，变量值超过 $k-1$ 时又从 0 开始。具体描述如下。

```
semaphore  empty = k,full = 0;
int in = 0,out = 0;
    Producer()
{
    生产一个物品;
    p(empty);
    物品存入 buf[in];
    in = (in + 1) % k;
    v(full);
}
Consumer()
{
    p(full);
    从 buf[out]取物品;
    out = (out + 1) % k;
    v(empty);
    消费;
}
```

### 4. 复杂 PC 问题的并发程序设计

**例 3-7**　假定有 $n$ 个生产者进程 $P_1$、$P_2$、…、$P_n$ 和 $m$ 个消费者进程 $C_1$、$C_2$、…、$C_m$，它们共享一个有 $k$ 个单元格的缓冲区 buf[$k$]，如图 3-12 所示。试用信号量机制实现它们的并发执行。

分析：与一般 PC 问题相比，这里增加了生产者和消费者的进程数，可以看出，生产者和消费者之间的同步关系没有变化。但是，在独立的生产者进程之间，可能有两个或两个以上的进程在各自生产了一个物品后，同时向缓冲区存放物品，这就需要控制，避免它们同时向一个单元格存入物品，所以，生产者进程之间存放物品的操作必须互斥执行。同样，为避免多个消费者进程同时从一个单元格取物品，消费者进程之间取

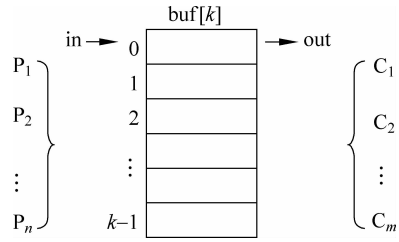



图 3-12　复杂 PC 问题

物品的操作也必须互斥执行。因此，得到复杂 PC 问题的并发程序设计，具体描述如下：

```
semaphore  empty = k,full = 0;
semaphore  mutex1 = 1,mutex2 = 1;
int in = 0,out = 0;
对于每一个生产者进程 Pi,i = 1,2,…,n,有
Pi(){
    生产一个物品;
    p(empty);
    p(mutex1);
    所生产的一个物品存入 buf[in];
    in = (in + 1) % k;
    v(mutex1);
    v(full);
}
对于每一个消费者进程 Ci,i = 1,2,…,m,有
Ci(){
    p(full);
    p(mutex2);
    从 buf[out]取一个物品;
    out = (out + 1) % k;
    v(mutex2);
    v(empty);
    消费;
}
```

### 3.6.9 读者/写者问题

读者/写者问题是另一个经典同步问题，在文件系统或多用户的数据库系统中，许多数据的操作具有与此相似的情况。

**例 3-8** 问题：假设有一个写者进程 Writer 和若干个读者进程 Reader，它们共享一组数据，写者进程 Writer 对数据进行写操作（如修改、删除、添加等），读者进程 Reader 对数据进行读操作，规定：①写操作与任一读操作之间，必须互斥执行；②多个读操作可以同时进行。如何用信号量机制实现它们的并发执行？

首先，分析如下并发程序设计。

```
semaphore ws = 1;
Writer(){
    p(ws);
    写操作;
    v(ws);
}
对于任一个 Reader 进程
Reader(){
    p(ws);
    读操作;
    v(ws);
}
```

上述并发程序设计可以实现问题中所提出的要求吗？可以看出，上述设计中，把“写操作”和“读操作”作为临界区，并定义互斥信号量 ws，因此，可以满足问题中的第①项规定，即写操作与任一读操作的互斥执行。但是，在这样的控制后，读者进程的读操作之间也被限制了，两个或多个读者进程不能同时执行读操作，所以，不满足问题中的第②项规定。

经过分析后发现，读者/写者问题虽然是互斥问题，但有其特殊性，这种特殊性表现在并不是一组进程全部都需要互斥执行，只有写者进程与任一读者进程之间要互斥执行，而读者进程之间可以同时进行读操作，不需要互斥执行。那么，如何处理这种特殊的互斥关系？

首先，分析读者进程开始运行（到来）的情况。当一个读进程开始运行时，如果这时有其他读者进程正在执行读操作，那么，可以推断得知，肯定没有写者进程在执行中，所以可以执行其读操作。这样，对第一个读者进程的运行控制成为解决问题的一个关键，如果第一个读者进程能够执行读操作，在其执行期间，后续的读者进程也可以执行读操作；同理，更多的读者进程开始运行时，只要有一个读者进程在执行读操作，那么，其他的读者也都可以跟着进行读操作，如果第一个读者进程不能执行读操作，那么，其他的读者进程也不能执行读操作。

其次，分析读者进程离开（结束）的情况。当一个读者进程完成他的读操作后，如果当前还有其他读者进程正在进行读操作，那么，他不能唤醒写者进程执行写操作，只须简单地离开即可，这样，对最后一个读者进程的离开控制成为关键。

所以，读者/写者问题就转化为如何解决写者进程与第一个读者进程和最后一个读者进程的互斥关系，为此，需要引入一个辅助变量表示当前执行读操作的读者进程的个数，以便区别第一个和最后一个。新来一个读者进程时，对这个变量执行加 1 操作，读者进程执行结束离开时，对这个变量执行减 1 操作。因为多个读者进程可能同时到来或离开，或者一个读者进程正到来时，另一个读者进程正要离开，所以对辅助变量的加 1、减 1 操作要互斥执行。

读者/写者问题的并发程序设计描述如下。

```
semaphore mutex = 1,ws = 1;
int readers = 0;
Writer()
{
    p(ws);
    写操作;
    v(ws);
}
```

对于任一个 Reader 进程

```
Reader()
{
    p(mutex);
    readers = readers + 1;
    if (readers == 1) p(ws);
    v(mutex);
    读操作;
    p(mutex);
    readers = readers - 1;
    if (readers == 0) v(ws);
    v(mutex);
}
```

进一步分析发现，在上述的处理读者/写者进程的并发程序设计中，读者进程相对较主动，或者说具有较高优先级。写者进程可能出现“饥饿”现象，因为只要有一个读者进程在执行读操作而未结束，后续的新读者进程就都可以进入执行读操作，意味着此时写者进程的写操作将一直被延迟，甚至无期限地等待。

在读者/写者问题中，为了具有更好的公平性，补充规定：当写者进程要执行写操作而阻塞后，现有正在执行读操作的进程可以继续，但新到来的读者进程要等待。上述的并发程序设计如何修改？

改进后的读者/写者问题的并发程序设计描述如下。

```
semaphore mutex = 1, ws = 1, mutex0 = 1;
int readers = 0;
Writer()
{
    p(mutex0);
    p(ws);
    写操作;
    v(ws);
    v(mutex0);
}
```

对于任一个 Reader 进程

```
Reader()
{
    p(mutex0);
    //虚拟一个申请操作
    v(mutex0);
    p(mutex);
    readers = readers + 1;
    if (readers == 1) p(ws);
    v(mutex);
    读操作;
    p(mutex);
    readers = readers - 1;
    if (readers == 0) v(ws);
    v(mutex);
}
```

需要指出，上述的并发程序设计，虚拟一个读者进程的“申请操作”，通过读者进程的这个“申请操作”与写者进程的写操作的互斥关系，实现读者/写者进程执行的公平性。另外，其中“v(mutex0)；”在唤醒 mutex0 对应等待队列的一个进程时，选择最先进入等待队列的一个进程。

思考：以上的读者/写者问题中，只有一个写者进程，如果有多个写者进程，上述的并发程序设计仍适用吗？

### 3.6.10 标志位机制

本节进一步讨论进程的互斥关系。从软件角度，介绍几个与程序设计方法相关的互斥

算法，掌握这些算法的设计思想对提高编程能力有很大的帮助。它们的共同点是在算法中设置了标志状态的变量，所以统称为标志位机制。

### 1. 严格轮转互斥算法

假定有 $n$ 个进程，它们分别是 $P_0$、$P_1$、…、$P_{n-1}$，严格轮转(Strictly Alternate)互斥算法(也简称严格轮转算法)实现这 $n$ 个进程互斥的思想是：定义一个用于表示轮转的整型变量 turn，初值 turn＝0。当 turn＝$i$ 时，进程 $P_i$ 进入临界区执行，$P_i$ 在离开临界区时设置 turn＝(turn＋1)％$n$。算法描述如下。

```
int turn = 0;
Pi()(i = 0,1, …, n - 1)
{
    …
    while(turn!= i);
    临界区;
    turn = (turn + 1) % n;
    …
}
```

该算法可以实现进程互斥关系，但算法有如下几个严重的不足。

(1) 算法的初始状态中，进程数 $n$ 固定。

这不符合并发执行的随机性，在多用户多任务环境下，用户数、进程数具有不确定性，算法初始就强制规定互斥的并发进程数，大大降低了算法的灵活性。

(2) 算法的初始状态中，规定了进程进入临界区执行的顺序，不满足临界区管理准则(1)。

事先规定一个顺序，各进程轮流地进入临界区执行，严重影响临界资源的利用率。因为在 turn 不等于 $i$ 时，即使此时没有其他进程在临界区执行，进程 $P_i$ 也不能进入临界区执行。也就是说，可能出现临界资源当前是空闲状态，要申请使用临界资源的进程却得不到它而等待，即忙等待(Busy Waiting)。

(3) 单个进程的崩溃会导致算法不能工作。

如果算法中某一个进程因故障等原因而异常终止，则其他进程将永久地等待。

尽管严格轮转互斥算法存在不足，但是，在一些特殊应用中，特别是在线程的应用中(线程概念详见本章 3.8 节)，如果正好满足算法中的条件，例如，一个进程的几个互斥(同步)线程中，线程数固定，而且它们的执行顺序事先按任务要求有严格的规定，这时采用严格轮转算法不失是一种好的选择，因为算法在应用程序中就可以实现，不需要操作系统的支持。

### 2. Dekker 互斥算法

在严格轮转算法中，一个进程不能连续两次进入临界区执行，存在忙等待的处理器浪费现象。这是因为算法只由变量 turn 登记当前是哪个进程可以进入临界区，对于其他进程是否想进入临界区执行却置之不理。

如果进一步登记进程是否要进入临界区执行的状态信息，是否可以摆脱严格轮转的限制呢?

**尝试 1** 以两个互斥进程为例，每个进程都有一个状态信息 flag：flag＝true 表示在临界区执行，flag＝false 表示不在临界区执行。特别地，flag[0]用于登记进程 $P_0$ 的状态，

flag[1]用于登记进程 $P_1$ 的状态。在一个进程要进入临界区执行之前，先检查对方进程的状态信息，以决定是否可以进入执行，一个进程在进入临界区执行时先置自己的状态信息为true，退出临界区时再置为 false。尝试 1 的算法如下。

```
boolean flag[2] = {false,false};
P0()
{
…
    while(flag[1] == true);
    flag[0] = true;
    临界区;
    flag[0] = false;
    …
}
P1()
{
    …
    while(flag[0] == true);
    flag[1] = true;
    临界区;
    flag[1] = false;
    …
}
```

实际上，这个修改算法与加锁机制类似，在 3.6.5 节已经分析过，软件的加锁机制不能实现互斥，不满足临界区管理准则(2)，所以，上述算法不能实现互斥关系，这种尝试不成功。

在尝试 1 算法中，先检查状态信息，然后设置状态信息，造成不能实现互斥。如果改变两个操作的顺序会如何呢？

**尝试 2** 先设置进程的状态信息，然后检查对方进程的状态信息，尝试 2 的算法描述如下。

```
boolean flag[2] = {false,false};
P0()
{
    …
    flag[0] = true;
    while(flag[1] == true);
    临界区;
    flag[0] = false;
    …
}
P1()
{
    …
    flag[1] = true;
    while(flag[0] == true);
    临界区;
    flag[1] = false;
    …
}
```

可以发现，尝试 2 算法可以实现 $P_0$、$P_1$ 两个进程的互斥。

但是，如果进程 $P_0$ 先执行，并在执行“flag[0]＝true；”后，处理器暂停 $P_0$ 的执行，而转向执行 $P_1$，那么 $P_1$ 在执行到其中的“while(flag[0]＝＝true)；”时，$P_1$ 的“while(flag[0]＝＝true)；”循环语句无法结束，之后，处理器在接着继续执行 $P_0$ 时，也进入“while(flag[1]＝＝true)；”循环也无法结束。

所以，尝试 2 算法不满足临界区管理准则(1)且造成进程的死锁(详见 4.4 节)。

**Dekker 算法** 借鉴尝试 2 算法和严格轮转算法，修改得到 Dekker 算法，实现 $P_0$、$P_1$ 两个进程的互斥关系。Dekker 算法思想如下。

对于进程 $P_0$ 要求进入临界区时，执行如下操作。

(1) 设置状态信息 flag[0]＝true，表示申请进入临界区执行。

(2) 检查对方进程 $P_1$ 状态信息，如果 $P_1$ 没有申请临界区执行(flag[1]＝false)，则可以进入临界区，转(5)。

(3) 在对方进程 $P_1$ 也在申请时，再检查严格轮转算法中的 turn，如果 turn 指示自己(进程 $P_0$)可以执行，则继续检查对方进程($P_1$)的状态，即转(2)。

(4) 此时，对方进程 $P_1$ 也在申请，且严格轮转算法中的 turn 指示进程 $P_1$ 可以执行。则进程 $P_0$ 暂时取消申请，置 flag[0]＝false，并循环检查 turn，直到 turn 指示自己(进程 $P_0$)可以执行，再置 flag[0]＝true，继续检查对方进程($P_1$)的状态，即转(2)。

(5) 执行临界区代码。

(6) 置 turn＝1，flag[0]＝false。算法结束。

对于进程 $P_1$ 要求进入临界区执行时的操作如下。

(1) 设置状态信息 flag[1]＝true，表示申请进入临界区执行。

(2) 检查对方进程 $P_0$ 状态信息，如果 $P_0$ 没有申请临界区执行(flag[0]＝false)，则可以进入临界区，转(5)。

(3) 在对方进程 $P_0$ 也在申请时，再检查严格轮转算法中的 turn，如果 turn 指示自己(进程 $P_1$)可以执行，则继续检查对方进程($P_0$)的状态，即转(2)。

(4) 此时，对方进程 $P_0$ 也在申请，且严格轮转算法中的 turn 指示进程 $P_0$ 可以执行。则进程 $P_1$ 暂时取消申请，置 flag[1]＝false，并循环检查 turn，直到 turn 指示自己(进程 $P_1$)可以执行，再置 flag[1]＝true，继续检查对方进程($P_0$)的状态，即转(2)。

(5) 执行临界区代码。

(6) 置 turn＝0，flag[1]＝false。算法结束。

实现 $P_0$、$P_1$ 两个进程的互斥的 Dekker 算法描述如下。

```
boolean flag[2] = {false,false};
int turn = 0;
P0()
{
    …
    flag[0] = true;
    while(flag[1] == true){
      if(turn == 1){
           flag[0] = false;
           while(turn == 1);
```

```
            flag[0] = true;
          }
        }
        临界区;
        turn = 1;
        flag[0] = false;
        …
}
P1()
{
    …
    flag[1] = true;
    while(flag[0] == true){
      if(turn == 0){
            flag[1] = false;
            while(turn == 0);
            flag[1] = true;
        }
      }
      临界区;
      turn = 0;
      flag[1] = false;
      …
}
```

### 3. Peterson 互斥算法

Dekker 算法可以实现进程互斥关系，但代码略显复杂，在此基础上，1981 年 Peterson 提出一个代码简单、设计思想非常巧妙的互斥算法，即著名的 Peterson 算法。

两个进程的 Peterson 算法思想是：一个进程在进入临界区之前要进行检查，如果发现对方在申请且轮到对方执行，则等待。也就是说，如果只有一个进程要求进入临界区执行，则允许其进入临界区，如果两个进程同时要求进入临界区执行，则让对方先执行，由最后一个得到让行的进程进入临界区执行。

实现 $P_0$、$P_1$ 两个进程互斥的 Peterson 算法如下。

```
boolean flag[2] = {false,false};
int turn = 0;
P0()
{
    …
    flag[0] = true;
    turn = 1;
    while(flag[1] = true && turn == 1);
    临界区;
    flag[0] = false;
    …
}
P1()
{
```

```
    …
    flag[1] = true;
    turn = 0;
    while(flag[0] = true && turn == 0);
    临界区;
    flag[1] = false;
    …
}
```

这里所介绍的 Peterson 算法，实现两个进程互斥，Peterson 算法也可以用于 $n(n>2)$ 个进程的互斥，作为本章习题之一，留给读者完成。

以上介绍的标志位机制，虽然是由软件实现的，但要求进程数已知，且进程之间共享标志变量，因此，不能用于应用进程之间的互斥，只能在操作系统内核模块之间，或者同一进程的几个线程之间使用。线程技术已经在广泛地应用之中，所以，学习、掌握标志位机制的思想和方法对提高编程能力有很大帮助。

### 3.6.11　管程机制

虽然信号量机制是最经典、应用最广泛的一种同步机制，但是在 3.6.8 节的生产者/消费者问题等并发程序设计应用中发现，连续多个信号量 p 操作的顺序非常重要，不合理的 p 操作顺序可能导致进程死锁，所以，应用信号量机制实现并发控制时要特别小心。另外，信号量机制中 p 操作和 v 操作分散在各个并发进程的程序中，不符合面向对象程序设计的思想。

霍尔(Hoare，1974)和汉森(Brinch Hansen，1975)提出了一种高级同步机制，称为管程(Monitor)。一个管程是实现并发控制的一组变量和过程(或函数)组成的一个抽象数据类型，一个管程组成一个特殊的模块或软件包，供其他过程或函数调用。管程是在程序设计语言一级的同步机制。管程的数据结构和过程由程序员设计、实现，因此称为高级同步机制。管程充分体现了面向对象的程序设计思想，使并发控制更加灵活。

本节介绍霍尔的管程思想和并发控制的应用。

#### 1. 管程的结构

管程结构描述如下。

```
monitor monitor_name {
    variable declarations;
    procedure proc_1(…){
          …
        }
    procedure proc_2(…){
          …
        }
    procedure proc_n(…){
          …
        }
    init(){
```

```
        monitor's initialization
    }
}
```

可以看出，一个管程由若干内部变量和一组过程组成，其中一个可选的、外部不可使用的特殊过程 init()，用于初始化管程，其他过程可供外部过程调用。

一个管程定义哪些变量和过程取决于实际应用需求和程序员的程序设计。

### 2. 管程特性

管程作为一种用于并发控制的特殊数据类型，具备如下 3 个特性。

(1) 共享性

一个管程可以供多个过程调用，具体地说，管程内部定义的过程可以供管程之外的其他过程或函数调用。

(2) 互斥性

管程是一种临界资源，在多个外部过程或函数调用管程的过程时，它们要互斥地访问同一个管程。即对于一个管程，如果当前有一个过程调用该管程的一个过程时，其他过程不得调用该管程的任何一个过程。

所以，如果将临界资源的使用设计为管程，则容易实现互斥关系。

(3) 安全性

管程可以用来表示资源，对资源的操作定义为管程内部的一组过程，这样，把对一个资源的访问操作集中在一个管程中，为保证资源的安全性建立基础。

### 3. 条件变量与管程

从以上看出，管程可以用于实现进程的互斥关系，那么，管程如何实现进程同步关系？

在管程的内部变量中，定义一类特殊的变量，称为条件变量(Condition Variables)，一个条件变量对应一个等待队列，程序员可以利用这个等待队列，在条件不满足时将调用进程(线程)加入对应的条件变量等待队列，而在合适的时候再将其从等待队列中唤醒。

为此，条件变量还需要两个操作，定义为：wait()和 signal()。

wait()操作将调用进程(线程)阻塞，同时归还管程，而 signal()操作将被 wait()操作阻塞的进程(线程)唤醒，如果没有阻塞进程(线程)，则 signal()是个空操作。

**例 3-9** 用于管理单资源的条件变量与管程设计如下。

```
monitor monitor_name {
    boolean busy;
    condition nobusy;
    procedure acquire(){
        if(busy) nobusy.wait();
        busy = true;
        }
    procedure release(){
         busy = false;
         nobusy.signal();
        }
```

```
…
init(){
      busy = false;
     }
}
```

进程调用管程的过程如下。

```
…
monitor_name.acquire();
临界区(含有调用管程 monitor_name 其他过程的代码);
monitor_name.release();
…
```

wait()和 signal()是在管程的过程中使用的,这样,管程机制面临细节上的一个重要问题:当一个进程 P 执行管程一个过程中的 signal()操作时,如果对应的条件变量的等待队列中有一个进程 Q 被唤醒。那么,因为 Q 是之前执行管程一个过程的 wait()操作而阻塞,现在被唤醒了,它当前处于调用管程的一个过程中,而 P 执行 signal()操作后也处于管程的一个过程中,P、Q 同时执行管程的过程,这与管程的互斥特性冲突。因此,管程机制需要进一步规定,来解决这个问题。

有两种基本的规定:

(1) P 等待,直到 Q 完成当前的管程操作,或等待另一个条件变量;

(2) Q 等待,直到 P 完成当前的管程操作,或等待另一个条件变量。

霍尔提倡使用规定(1),而汉森采取两者的折中,即对于管程中调用 signal()的过程,signal()操作必须是过程的最后一个操作,这样进程 P 在执行 signal()操作后,立即自动完成管程的操作。

关于条件变量与管程的应用,将在 3.8.5 节结合 Java 程序设计语言进一步介绍。

#### 4. 信号量机制与管程

霍尔证明信号量机制可以由管程机制实现,反过来,管程机制也可以由信号量机制实现。下面,介绍霍尔应用信号量机制实现的单资源管理的管程设计方法和例子。

管程内部定义互斥信号量 mutex(初值为 1)控制各进程调用管程时的互斥关系。

管程内部定义信号量 urgent(初值为 0),用于控制执行 signal()操作的进程,即执行 signal()操作时通过 p(urgent)阻塞调用进程。

管程内部定义一个整型变量 urgentcount 表示等待 urgent 的进程数,初值 urgentcount=0。

定义一个抽象数据类型,充当条件变量的作用。其中定义信号量 consem,初值为 0,用于控制等待资源的进程,即当资源处于忙(busy)时,通过 p(consem)阻塞调用进程;另外,定义一个整型变量 condcount 表示等待 consem 的进程数,初值 condcount=0。用信号量机制实现该抽象数据类型的两个方法:wait()和 signal()。

信号量机制实现单资源管理的管程结构设计如下。

```
monitor monitor_name {
    semaphore mutex = 1;
    semaphore urgent = 0;
```

```
int urgentcount = 0;
boolean busy = false;
class cond{
    semaphore condsem = 0;
    int condcount = 0;
    procedure wait(){
        condcount = condcount + 1;
        if(urgentcount > 0)
            v(urgent);
        else
            v(mutex);
        p(condsem);
        condcount = condcount - 1;
    }
    procedure signal(){
        urgentcount = urgentcount + 1;
        if(condcount > 0){
            v(condsem);
            p(urgent);
        }
        urgentcount = urgentcount - 1;
    }
}
cond nobusyCond;
procedure entry(){
    p(mutex);
}
procedure exit(){
    if (urgentcount > 0)
        v(urgent);
    else
        v(mutex);
}
procedure acquire(){
    if(busy)
        nobusyCond.wait();
    busy = true;
    }
procedure release(){
    busy = false;
    nobusyCond.signal();
    }
}
```

**例 3-10** 应用管程实现复杂 PC 问题的并发程序设计。

假定缓冲区容量为 $k(k>1)$，管程复杂 PC 问题的并发程序设计如下。

```
monitor monitor_PC {
    semaphore mutex = 1;
    semaphore urgent = 0;
    int urgentcount = 0;
```

```
class cond{
    semaphore condsem = 0;
    int condcount = 0;
    wait(){
          condcount = condcount + 1;
          if(urgentcount > 0)
             v(urgent);
          else
             v(mutex);
          p(condsem);
          condcount = condcount − 1;
    }
    signal(){
          urgentcount = urgentcount + 1;
          if(condcount > 0){
             v(condsem);
             p(urgent);
          }
          urgentcount = urgentcount − 1;
    }
}
cond emptyCond,fullCond;
int count = 0;
int in = 0,out = 0;
BoundedBuffer < Item > buf[k];
procedure entry(){
    p(mutex);
}
procedure exit(){
     if (urgentcount > 0)
          v(urgent);
     else
          v(mutex);
}
procedure append(Item x){
    if(count == k)
          emptyCond.wait();
    count = count + 1;
    buf[in] = x;
    in = (in + 1) % k;
    fullCond.signal();
    }
procedure remove(){
    Item x;
    if(coun == 0)
        fullCond.wait();
    count = count − 1;
    x = buf[out];
    out = (out + 1) % k;
    emptyCond.signal();
    return x;
```

```
        }
    }
```

生产者进程：

```
…
生产一个物品 x;
monitor_PC.entry();
monitor_PC.append(x);
monitor_PC.exit();
…
```

消费者进程：

```
…
monitor_PC.entry();
x = monitor_PC.remove();
monitor_PC.exit();
…
```

在上述的管程设计中，限制了生产者进程与消费者进程的互斥关系。

**例 3-11** 应用管程实现 3.6.9 节读者/写者问题的并发程序设计。

管程实现读者/写者问题的并发程序设计如下。

```
monitor monitor_WriterReader {
    semaphore mutex = 1;
    semaphore urgent = 0;
    int urgentcount = 0;
    boolean writing = false;
    class cond{
        semaphore condsem = 0;
        int condcount = 0;
        procedure wait(){
              condcount = condcount + 1;
              if(urgentcount > 0)
                  v(urgent);
               else
                  v(mutex);
               p(condsem);
               condcount = condcount - 1;
        }
        procedure signal(){
            urgentcount = urgentcount + 1;
            if(condcount > 0){
                v(condsem);
                p(urgent);
            }
            urgentcount = urgentcount - 1;
         }
        procedure getCondcount(){
            return condcount;
        }
```

```
    }
    cond readerCond,writerCond;
    int rCount = 0;
    procedure entry(){
        p(mutex);
    }
    procedure exit(){
        if (urgentcount > 0)
            v(urgent);
        else
            v(mutex);
    }
    procedure acquireRead(){
        if(writing || writerCond.getCondcount()> 0)
            readerCond.wait();
        rCount = rCount + 1;
        readerCond.signal();
        }
    procedure releaseRead(){
        rCount = rCount - 1;
        if(rCount == 0)
            writerCond.signal();
        }
    procedure acquireWrite(){
        if(writing || rCount > 0)
            writerCond.wait();
        writing = true;
        }
    procedure releaseWrite(){
        writing = false;
        if(readerCond.getCondcound > 0)
            readerCond.signal();
        else
            writerCond.signal();
        }
}
```

写者进程：

```
…
monitor_WriterReader.entry();
monitor_WriterReader.acquireWrite ();
写操作;
monitor_WriterReader.releaseWrite ();
monitor_WriterReader.exit();
…
```

读者进程：

```
…
monitor_WriterReader.entry();
monitor_WriterReader.acquireRead ();
```

```
读操作;
monitor_WriterReader.releaseRead ();
monitor_PC.exit();
…
```

上述的并发设计中，允许有多个写者进程，写者进程之间的“写操作”是互斥关系。

## 3.7 进程通信

本节介绍进程管理的第三个功能——进程通信。多道程序设计的并发执行不仅提高了资源的利用率，同时为多个进程的任务协作提供了可能，进程之间的任务协作需要进程通信功能的支持。

### 3.7.1 进程通信的概念

#### 1. 什么是进程通信

两个或多个进程之间交换数据的过程称为进程通信，其中提供数据的一方称为发送进程，得到数据的一方称为接收进程。

#### 2. 进程通信类型

进程通信分为两种类型：低级通信和高级通信。

低级通信是指操作系统内核程序之间的通信，交换的数据量较小，且交换的数据用于控制进程的执行。信号量机制就是一种低级通信。

高级通信是指应用程序之间的通信，交换的数据量可以很大，且交换的数据是接收进程的处理对象。

本书中，没有特别说明，进程通信是指高级通信，即应用程序之间的数据交换。

#### 3. 为什么需要进程通信

在多道程序环境下，多个进程共享同一个内存，表面上，进程之间交换数据似乎很简单，其实不然，操作系统需要提供专门的进程通信机制，其主要原因如下。

(1) 任务协作

为了提高并行程度，往往把一个任务称为主任务，分解成几个子任务，一个子任务对应一个进程，通过这些进程的并发执行，共同协作完成主任务的功能，任务协作的过程中通常都需要交换数据。

(2) 进程的独立性

进程的特征之一是进程独立性，这是操作系统为了方便管理所做出的一种限制，一个进程不能访问另一个进程的数据或代码，以保证进程之间不会相互干扰，但也因此造成进程之间无法直接交换数据，而需要借助进程通信机制来实现。

#### 4. 进程通信的可行性

操作系统既然规定了进程的独立性，使进程之间无法直接交换数据，那么，进程之间这

种交换数据的要求可以实现吗？虽然进程之间不能直接交换数据，但是，这种要求还是可以实现的。例如，如图 3-13 所示的两种方案。

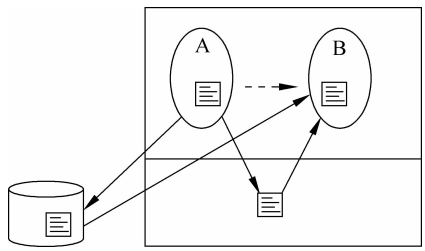



图 3-13　进程通信方案

假定进程 A 需要把一组数据提交给进程 B。一种方案是，利用磁盘等辅助存储器，发送、接收双方经过事先的约定，发送进程 A 把要交换的数据写入辅助存储器的指定位置，接收进程 B 从该位置读取数据。这种方案实现简单，只需操作系统的文件系统即可，但通信过程需要 I/O 操作。

另一种方案是，利用内核运行在核心态的特点，发送进程 A 通过内核程序将数据写入内核空间指定区域，接收进程 B 通过另一组内核代码，内核从指定区域读取数据，并写入接收进程 B 的地址空间，从而实现数据从进程 A 交换到进程 B。这种方案需要操作系统通过专门的系统调用来实现，数据交换的速度快。

## 3.7.2　进程通信方式

进程通信方式是指进程通信的具体实现方法，主要有以下几种。

### 1. 共享存储区通信

共享存储区通信的基本思想是：建立一个共享存储区域，通信时，把这个区域地址映射到发送进程的地址空间，使得发送进程可以访问这个共享存储区域，同时这个共享存储区域地址也可以映射到接收进程的地址空间，使得接收进程也可以访问该区域。

在这种通信方式中，通信机制本身只提供共享存储区域的申请、映射等基本操作，对共享区域的读、写操作等的同步、互斥控制，程序员则要根据具体数据的应用要求，对发送进程和接收进程进行合理协调。

### 2. 消息缓冲通信

消息缓冲通信是最基本的进程通信，为其他的进程通信方式的发展建立基础，其思想是：每个进程都对应一个消息缓冲区队列，并提供发送和接收两个操作，发送进程利用发送操作，将数据组织在消息缓冲区中，并将消息缓冲区加入接收进程的消息缓冲区队列中，接收进程将来通过接收操作，从自己的消息缓冲区队列中取出消息缓冲区，从而得到发送进程的数据。本节后面将进一步介绍消息缓冲通信的设计、实现。

### 3. 信箱通信

信箱通信是应用最广泛的进程通信，在后面也将另做介绍。

#### 4. 管道通信

管道(pipe)通信的基本思想是利用文件系统的功能，通信双方的进程共享同一个文件，发送进程向文件中写数据，接收进程从文件中读数据，打开的共享文件类似一段“管道”，数据从一端流向另一端。

### 3.7.3 消息缓冲通信的设计和实现

下面介绍消息缓冲通信的设计、实现。

#### 1. 消息缓冲通信设计

消息缓冲通信的设计主要包括数据结构设计和发送、接收操作的设计，其中数据结构主要有消息缓冲区的结构、PCB 的通信参数结构。

(1) 消息缓冲区结构

把要发送的数据称为消息，用于存放消息的内存区域称为消息缓冲区，消息缓冲区的结构至少由以下 4 个方面组成。

① 发送进程标识(pid)：登记发送进程的 pid。

② 正文大小(size)：进程要交换的数据称为正文，按字节计算的字符数量。

③ 正文(data)：存储发送进程提交给接收进程的数据，这是通信的主要内容。

④ 向下指针(Next)：一个消息缓冲区作为队列的一个结点，指针指示了在消息缓冲区队列中的下一个结点。

(2) PCB 的通信参数结构

每个进程的进程控制块 PCB 中，包含以下 3 个内容。

① 消息缓冲区队列(mq)：用于组织到来的信息缓冲区。

② 互斥信号量(mutex)：消息缓冲区队列是一个链表，链表结点的添加或删除都需要互斥执行，因为在这种通信机制中，可能有多个进程同时向一个进程发送数据，或者一个进程在发送消息期间接收进程也可能接收消息。

③ 同步信号量(msg)：接收进程依赖于发送进程，由于链表没有结点个数的限制，发送进程可以不受接收进程的限制，通信双方是一种简单同步关系。

(3) 发送操作和接收操作

发送操作设计如下。

格式：send(dest,&mptr)

参数：dest 为接收进程 pid，mptr 为发送区地址，属于发送进程的地址空间，发送区结构与消息缓冲区结构类似，只是发送区不含向下指针，发送区数据由发送进程初始化。

功能：申请一个新的消息缓冲区，把 mptr 指示的发送区数据复制至消息缓冲区，并把消息缓冲区作为一个结点加入接收进程 dest 的消息缓冲区队列。

接收操作设计如下。

格式：receive(&mptr)

参数：mptr 为接收区地址，属于接收进程的地址空间，用于接收到来的消息，接收区结构与发送区结构一样。

功能：从接收进程 PCB 对应的消息缓冲区队列中移出一个消息缓冲区，并把消息缓冲区的数据（除向下指针）复制至接收区，并归还消息缓冲区。

### 2. 消息缓冲通信实现

如图 3-14 所示，描述了消息缓冲通信的实现，这是一个典型的简单同步关系的例子。

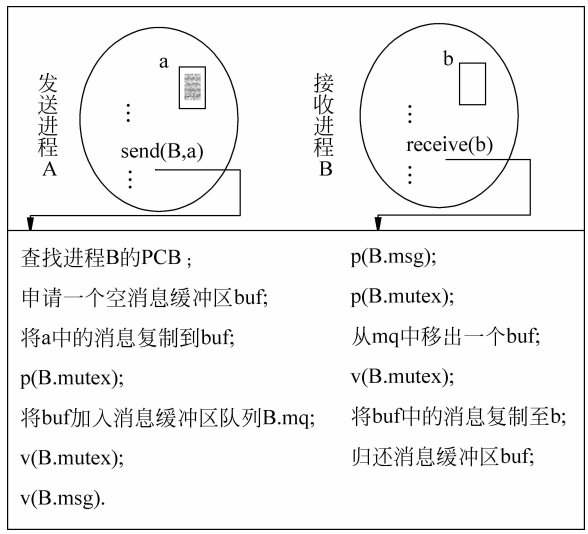



图 3-14　消息缓冲通信的实现

从实现过程可以看出：

（1）消息缓冲通信是一种直接通信

“直接”的含义是发送进程在调用 send()时，首先要查找接收进程的 PCB，此时，如果接收进程尚未被创建，则通信不成功。把消息缓冲区队列建立在接收进程内部，通信缺乏灵活性。

（2）只能应用在同一台计算机

由于 send()要把消息缓冲区直接加入接收进程对应的消息缓冲区队列中，因此，发送进程和接收进程必须在同一内存中执行。所以，消息缓冲通信只能用于同一台计算机的进程通信。

## *3.7.4　UNIX 消息队列通信

UNIX 的消息队列通信机制，把消息缓冲区队列（以下简称消息队列）从 PCB 中独立出来，通信双方自由约定建立一个或多个消息队列，也可以使用其他进程已经建立的消息队列，消息格式也可以扩展，为同一台计算机上的进程通信提供了一种灵活的方法。

下面简要介绍 UNIX 关于消息队列的 4 个系统调用，并给出一个应用实例。

### 1. 消息队列系统调用

4 个系统调用是分别如下。

(1) msgget(key_t key, int msgflg)

功能：返回一个与参数 key 对应的消息队列的标识符 qid。

参数：key 是一个由通信双方事先约定的值。消息队列是内核的数据结构，在内核中每一个消息队列都有唯一的队列标识符 qid，当内核新建消息队列时，自动为新队列生成一个 qid，将 key 与 qid 建立对应关系。之后，程序员在发送或接收程序中可以通过 key 获取 qid，进行发送或接收操作。

当 key=IPC_PRIVATE 时内核将新建立一个消息队列；当与 key 对应的消息队列不存在且 msgflg&IPC_CREAT 非零时，内核也新建立一个消息队列；当与 key 对应的消息队列存在且 msgflg 包含的用户访问权限与对应的消息队列的权限匹配时，返回 key 对应的 pid。消息队列的访问权限语义与文件系统的访问权限的语义相同(可参看 6.8.2 节关于存取控制表的介绍)。

当新建一个消息队列时，内核建立并初始化描述消息队列状态的数据结构 msqid_ds，其中包含：

msg_perm.cuid 和 msg_perm.uid 设置为当前用户标识符 uid；

msg_perm.cgid 和 msg_perm.gid 设置为当前用户组标识符 gid；

msg_perm.mode 的低 9 位设置为 msgflg 的低 9 位数据；

msg_qnum、msg_lspid、msg_lrpid、msg_stime 和 msg_rtime 均为 0；

msg_ctime 设置为当前系统时间；

msg_qbytes 设置系统限制的 MSGMNB。

返回：msgget()返回非负整数时成功，返回－1 表示错误。

(2) msgsnd(int msgqid, struct msgbuf *msgp, size_t msgsz, int msgflg)

功能：将从消息指针 msgp 指示的地址开始，长度为 msgsz 字节的数据作为消息，加入 msgqid 对应的消息队列中。

参数：msgqid 表示消息发送到哪个消息队列，这里是通过 msgget(key_t key, int msgflg)得到的队列标识符。struct msgbuf 是 UNIX 的消息格式，其基本结构定义如下：

```
struct msgbuf {
long mtype;
char mtext[1];
}
```

程序员可以扩展 msgbuf 的结构，只要保证 msgbuf 结构的第一项是 long mtype，其他内容程序员可以根据应用的要求进行扩展。

消息的长度 msgsz 的含义：msgsz 等于在 msgbuf 的基本结构的 mtext 的实际字符长度，如果程序员扩展了 msgbuf，则可以定义 msgsz=sizeof(struct msgbuf)－sizeof(long)。

msgflg 的作用是指示是阻塞发送还是非阻塞发送。

返回：返回值为 0 时发送正确，返回值为－1 时表示发送错误。

(3) msgrcv(int msgqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg)

功能：从指定队列中接收一个消息。

参数：msgqid 表示从哪个消息队列接收消息，这里是事先通过 msgget(key_t key, int

msgflg)得到的队列标识符,msgp 表示接收的消息要存放的位置,通信双方事先必须约定 msgbuf 的结构,只有这样,接收进程才能正确地分析消息中的各项数据,msgflg 用于指示是阻塞发送还是非阻塞发送。

msgtyp 用来指示所要接收的消息类型:当 msgtyp=0 时,接收指定队列中的第一个消息;当 msgtyp>0 时,接收指定队列中消息的 mtype 等于 msgtyp 的第一个消息(msgflg 不包含 MSG_EXCEPT 时);当 msgtyp<0 时,接收指定队列中消息的 mtype 小于或等于-msgtyp 的第一个消息。

程序员可以利用 msgrcv()中的参数 msgtyp 和消息中的 mtype,经过合理的设计就可以实现一个进程与多个进程通信。例如,进程 A 和 B 同时与进程 C 通信,在 A、B 和 C 共享一个消息队列时,可以规定:mtype=1 的消息为进程 C 的接收消息,mtype=进程 A 的 pid 的消息为进程 A 的接收消息,mtype=进程 B 的 pid 的消息为进程 B 的接收消息。这样,进程 A 和 B 在向 C 发送消息时,它们设置所发送的消息的 mtype=1,同时在消息中加上自己的进程标识符 pid;进程 C 用 msgtyp=1 接收队列中的消息,进程 C 在收到消息后,可以利用消息中的进程标识符 pid(即原发送进程的 pid),作为应答消息的 mtype,这样就可以保证应答消息能够被原发送进程接收。由此可见 UNIX 系统调用的灵活性。

msgflg 可以为 0 或以下 3 项的组合。

① IPC_NOWAIT——如果没有可接收的 msgtyp 类型消息且 msgflg 不包含 IPC_NOWAIT,则进程阻塞,直到期望消息到达,或者消息队列被取消或进程捕获一个 signal;如果消息队列中没有指定消息而 msgflg 包含 IPC_NOWAIT,则立即返回错误。

② MSG_EXCEPT——当 msgtyp 大于 0 时,读取消息队列中 mtyp 不等于 msgtyp 的第一个消息。

③ MSG_NOERROR——当队列中的消息长度超过 msgsz 时截去超出部分的数据。

返回:返回值大于或等于 0 时表示实际接收的消息字符数,返回值等于-1 表示接收错误。

(4) msgctl(int msgqid, int cmd, struct msgqid_ds * buf)

功能:对 msgqid 对应的消息队列,执行指定 cmd 的控制操作。

参数:msgqid 表示要执行控制操作的消息队列。控制操作 cmd 有以下 3 种取值。

① IPC_STAT——读取指定消息队列状态数据结构的当前值。

② IPC_SET——设置指定消息队列状态数据结构的部分数据,主要有 msg_ctime、msg_perm.uid、msg_perm.gid、msg_perm.mode(低 9 位)和 msg_qbytes 等。

③ IPC_RMID——立即删除指定的消息队列及其相关的状态数据结构,唤醒该队列的阻塞进程。

返回:返回值为 0 表示成功,返回值为-1 表示操作错误。

### 2. 消息队列应用实例

这里给出一个消息队列通信的应用实例的 C 语言源代码。

**例 3-12** 两个进程利用通信进行协作,实现简单的四则运算。一个进程称为客户进程(Client),其任务是从键盘上接收三个参数:两个操作数和一个运算操作符,然后把这三个参数发送给另一个进程,即服务器进程(Server);服务器进程接收客户进程的请求,并根据

提供的运算操作符，对两个操作数进行运算，服务器把运算的结果返回给客户。

（1）头文件

头文件 msg_mycs.h 定义扩展的消息结构及双方约定的 KEY。

```
#define KEY 1183
struct msgbuf {
    long mtype;
    int source_pid;
    double a,b;
    char opcode;
    double result;
    char return_msg[128];
}msg;
int msgqid;
int msgsize = sizeof(struct msgbuf) - sizeof(long);
```

通信双方的源程序应包括此头文件。

（2）服务器程序

源程序文件 server.c 实现简单的四则运算的服务，称为服务器（Server）。运行时，首先建立一个消息队列，然后执行一个循环：接收其他进程发送的请求消息、分析消息请求的服务类型、处理请求以及服务结果返回给请求的进程，然后回到循环顶部，接收一下请求。

服务器程序的源代码如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "msg_mycs.h"
main(){
    int i;
    extern cleanup();
    for(i = 0;i < 20;i++)
     signal(i,cleanup);
    msgqid = msgget(KEY,0777|IPC_CREAT);
    for(;msg.opcode!= 'q';){
     printf("server pid = %d is ready (msgqid = %d)…\n",getpid(),msgqid);
     msgrcv(msgqid,&msg,msgsize,1,0);          //约定服务器接收的消息类型 mtype = 1
         printf("server: receive from pid = %d\n",msg.source_pid);
         msg.return_msg[0] = '1';
         switch(msg.opcode){
         case '+':
         msg.result = msg.a + msg.b; break;
     case '-':
        msg.result = msg.a - msg.b; break;
     case '*':
        msg.result = msg.a * msg.b; break;
     case '/':
        if(msg.b!= 0)
          msg.result = msg.a/msg.b;
        else
```

```
            strcpy(msg.return_msg,"0.divide by 0.");
            break;
        default:
            strcpy(msg.return_msg,"0.EXIT by client user.");
            break;
        }
    if(msg.return_msg[0] == '1')
        printf(" %.2f %c %.2f = %.2f \n",msg.a,msg.opcode,msg.b,msg.result);
    msg.mtype = msg.source_pid;                    //返回给客户消息类型用客户进程 pid
    msg.source_pid = getpid();
    msgsnd(msgqid,&msg,msgsize,0);
    }
    printf("server exit by client pid = %d\n",msg.source_pid);
}
cleanup(){
    msgctl(msgqid,IPC_RMID,0);
    exit();
}
```

**注意**：上述源代码中“for(i=0;i<20;i++) signal(i,cleanup);”的作用是：设置软中断，当出现这些中断信号时，调用 cleanup()删除或撤销指定消息队列。

为什么要通过软中断删除消息队列？因为一个进程在建立一个消息队列后，该消息队列是在 UNIX 的内核，可以供多个进程共享。在一个进程完成后如果没有主动删除或撤销消息队列，内核在撤销进程时是不会自动删除的。而进程主动删除消息队列又很困难，因为进程可能随时以不同方式终止，程序员安排的删除消息队列的代码不一定会得到运行，如果这样，已经不再需要的消息队列就有可能仍然保留在内核，占用内核有限的资源，程序员必须要避免这种状况。一个有效的方法就是应用 UNIX 提供的软中断功能。通过设置软中断，当进程终止时产生软中断信号，中断处理程序调用 cleanup()删除或撤销指定消息队列。

(3) 客户端程序

源程序文件 client.c 通信中的另一方，称为客户(Client)。运行时，首先按双方约定的 KEY 获取服务器方建立的消息队列，然后运行一个循环：等待用户键盘输入一道四则运算的三个参数：操作数 a 和 b、操作符 opcode，并发送请求消息，紧接着调用 msgrcv()接收结果，假定当 opcode='q'时，双方通信过程结束。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "msg_mycs.h"
main(){
    struct msgbuf msg;
    int pid;
    msgqid = msgget(KEY,0777);
    pid = getpid();
    for(;msg.opcode!= 'q';){
        printf("a ( +- */) b = ? \na = ");
        scanf("%lf",&msg.a);
        printf("b = ");
```

```
            scanf("%lf",&msg.b);
            printf("opcode=(+,-,*,/,q for EXIT)");
        msg.opcode=getchar();
        while(msg.opcode=='\n')msg.opcode=getchar();        //忽略回车符(\n)
        if(msg.opcode=='+'||msg.opcode=='-'||'*'==msg.opcode||
                '/'==msg.opcode||'q'==msg.opcode){          //假定客户输入'q'键结束通信
            msg.source_pid=pid;
            msg.mtype=1;
            msg.return_msg[0]=0;
            msgsnd(msgqid,&msg,msgsize,0);
            msgrcv(msgqid,&msg,msgsize,pid,0);              //接收等于客户进程 pid 的消息
            printf("client: receive from pid=%d\n",msg.source_pid);
            if(msg.return_msg[0]=='1')
              printf("%.2f %c %.2f=%.2f\n\n",msg.a,msg.opcode,msg.b,msg.result);
            else
             printf("\n%s\n",msg.return_msg);
        }
    }
}
```

在这个例子中，一个四则运算的任务分解成两个子任务：客户端(Client)和服务器(Server)。客户端程序实现人机交互部分的代码，完成请求的提出和结果的处理，服务器程序负责实现具体的数据处理部分的代码，这种结构设计方法，人们在程序设计中经常采用，经过规范和发展，成为一种固定的结构模式，即 C/S 结构。

## 3.7.5 信箱通信的设计实现

信箱通信是一类应用最广泛的进程通信，其思想借鉴了日常生活中邮政系统的邮件投递方式。寄件人在邮件写好之后，注明收件人地址和姓名等信息，就可以投入就近邮局的任何一个邮箱；邮政工作人员定期收集、整理、分类后，邮件经过一次或多次的转发，到达了收件人所在地址的就近一个邮局，邮局工作员再把邮件投递到收件人的邮箱；收件人可以在自己的邮箱中取得邮件，这样信息就从寄件人传送到了收件人。

下面只针对单处理器的情况，介绍信箱通信的设计、实现。略去邮件的中间存储转发的过程，只考虑发送进程如何将邮件存入信箱，以及接收进程如何从信箱取出邮件。

### 1. 信箱结构

把进程之间交换的数据组织成信件。

信箱(Mailbox)是一个固定的存储区域，一个信箱由信箱头和信箱体两部分组成。信箱头包含信箱的描述、控制信息，主要内容如下。

信箱名(boxname)：信箱名称。

信箱标识符(bid)：系统在建立新的信箱时生成的唯一标识符。

信箱大小(size)：信格总数。

同步信号量(mailnum)：与信箱中信件数量相关的信号量。

同步信号量(freenum)：与信箱中空信格数量相关的信号量。

读互信号量(rmutex)：接收信件时的互斥信号量。

写互信号量(wmutex)：存入信件时的互斥信号量。

读信件指针(out)：当前可存入信件的信格地址。

存信件指针(in)：当前可读信件所在的信格地址。

还可以包括其他的信息。

除信箱头外，信箱的其余部分称为信箱体，信箱体是由若干个连续的称为信格的区域组成的缓冲区数组 buf[size]，一个信格存放一个信件，一个信件也只占用一个信格。

### 2. 信箱的实现

信箱定义后，还要有两个基本操作，即发送和接收操作。

(1) 发送操作

格式：send(dest, &mptr)。

参数：dest 为信箱标识符，mptr 为用户信件信息存入的地址。

功能：将 mptr 指示的用户信件，存入信箱 dest，如果当前信箱满，也就是当前没有空余的信格，则发送进程进入阻塞状态，直到有信件被读取后才可存入。

(2) 接收操作

格式：receive(addr,&mptr)。

参数：addr 为信箱标识符；mptr 指示信件接收后存入的位置。

功能：从 addr 对应的信箱中读取一个信件，并存入 mptr 指示地址区域。如果当前信箱为空，即信箱中没有信件，则接收进程进入阻塞状态，直到有信件存入后才可读取。

信箱通信的实现过程如下。

```
send(dest,&mptr)
{
    p(dest.freenum);
    p(dest.wmutex);
    dest.buf[dest.in] ← mptr;
    dest.in = (dest.in + 1) % dest.size;
    p(dest.wmutex);
    p(dest.mailnum);
}
receive(addr,&mptr)
{
    p(addr.mailnum);
    p(addr.rmutex);
    mptr ← addr.buf[addr.out];
    addr.out = (addr.out + 1) % addr.size;
    p(addr.rmutex);
    p(addr.freenum);
}
```

信箱通信方式可以用于计算机网络、分布式系统中不同计算机进程之间的通信，也称为消息传递通信方式。

消息传递通信方式的设计和实现要复杂得多，例如发送进程和接收进程的标识或寻址、

阻塞通信还是非阻塞通信、可靠还是不可靠通信等，以及发送进程和接收进程所在的计算机在硬件和软件上的差异等都要一一考虑。但是，消息传递通信方式是计算机网络的通信基础，在此基础上，发展起来的远程过程调用（Remote Procedure Call，RPC）技术，应用非常广泛。这里不一一展开，有兴趣的读者可以查阅计算机网络的相关文献。

## 3.8 线程

系统工作流程从程序的顺序执行发展到并发执行后，引用了进程的概念，实现对处理器的有效管理。通过进程之间的并发执行，发挥了硬件上处理器和设备的并行工作的能力，提高了资源的利用率。当一个进程因为 I/O 操作等原因导致处理器等待时，操作系统让本来要等待的处理器运行另一个进程。但是，随着研究和应用的不断深入，人们发现系统工作的基本单位的粒度还可以进一步细化，把进程细化为若干个线程，实现进程内部的并发执行。

### 3.8.1 线程的引入

一个进程因执行某一个特殊指令进入阻塞状态后，进程后续的指令都不能运行，即使这些指令与这个特殊的指令没有任何逻辑关系，也因进程的阻塞而不能运行。所以，在进程中，个别指令的执行可能影响整个进程的状态，如果能够把进程做进一步细化，使得个别指令的执行所带来的影响限制在进程中一个较小范围内部，进程的其他部分仍然可以运行，那么，将进一步提高处理器的效率。

#### 1. 什么是线程

关于线程的含义，有许多种不同的描述。这里线程描述为：把进程细化成若干个可以独立运行的实体，每一个实体称为一个线程（Thread）。

进程的细化取决于程序员的设计，程序员根据实际应用的需要，将一些可以单独执行的模块设计为线程。需要指出，引入线程后，“进程就是由线程组成的”这个观点不正确，因为有的进程不能够完全细化，总有一些模块无法独立于其他模块而运行；另外，即使有些模块可以独立执行，但也不一定都要设计为线程。因此，引入线程后，进程中仍有部分代码以原来的方式执行，只是其中的线程部分独立于进程而执行。

#### 2. 引入线程的目的

引入线程可以减小系统的基本工作单位粒度，其目的如下。

（1）实现进程内部的并发执行，提高并行程度

将进程细化，得到若干线程之后，线程与线程之间、线程与进程之间可以并发执行，实现进程内部实体间的并发执行，如果有多个处理器，线程之间可以并行执行。

特别是在计算机网络系统的应用中，在经过良好的设计后，将服务器进程细化为若干个线程，可以实现多用户的并行操作。因为服务器进程细化后，某一个用户的请求，由一个线程来处理，造成阻塞时只是该线程的阻塞，进程中其他的部分仍是活动的，还可以继续响应、处理其他用户的请求。如果服务器进程没有细化，那么，一个用户的请求处理造成进程阻塞

时，服务器对其他用户请求的响应、处理将被推迟。

(2) 减少处理器切换带来的开销

多个进程的并发执行，实现了处理器和设备的并行工作，当一个进程执行某一个特殊的操作(如I/O操作等)时暂时不能使用处理器，操作系统立即把它设置为阻塞状态，处理器分配给下一个进程，从而减少处理器的等待时间。但是由于进程的独立性，下一个运行的进程不能使用也不需要使用原来进程的资源，所以，操作系统需要把原来进程的现场保护到PCB中，避免受到下一个运行进程的影响，并保证原来的进程下一次得到处理器时能够接着继续运行。

引入线程后，线程之间的并发执行，同样可以减少处理器的等待时间，并且，一个线程阻塞后，处理器可以执行同一个进程的另一个线程，这个线程与原来的线程共享同一个进程的资源，因此，只需要保护原线程的少数几个寄存器和堆栈信息，即线程的现场信息比进程的要少得多，从而减少处理器切换带来的系统开销。

(3) 简化进程通信方式

并发执行方式不仅提高了资源的利用率，还为多任务的协作提供了可能。但是由于进程的独立性，进程间的任务协作必须借助进程通信机制来实现。引入线程以后，程序员可以把进程通信的双方设计为同一个进程的两个线程，这样就可以利用它们共享的进程地址空间来交换数据，从而简化了进程通信，加快了交换数据的过程。

### 3.8.2 线程与进程的关系

引入线程后，同一进程的线程之间共享该进程的地址空间。线程与进程的根本区别是：线程是处理器分配调度的基本单位，进程是其他资源(除处理器之外)分配的基本单位。另外，进程的地址空间是私有的，处理器在进程之间切换时现场的保护/恢复的开销比较大，同一进程的线程在处理器之内切换时现场的保护/恢复的开销比较小。

与进程一样，线程具有动态性，每一个线程都有生命期，具有一个从创建、运行到消亡的过程；线程也具有并发性，多个线程可以并发执行；线程也有三个基本状态：运行、就绪和阻塞；具有相互制约关系的线程之间也需要同步、互斥控制。

### 3.8.3 线程的类型

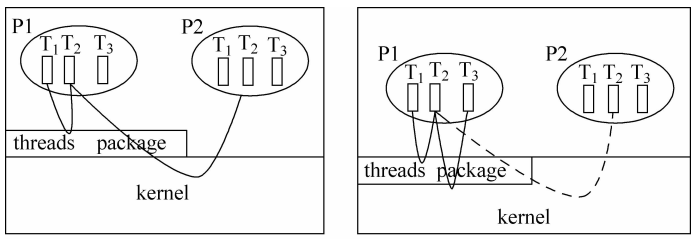
线程也具有动态性、并发性等特征，线程也需要同步、互斥等控制。管理、控制线程的模块称为线程包(Threads Package)，根据线程包的不同实现方式，将线程分为用户级线程(User Thread)和系统级线程(Kernel Thread)。

#### 1. 用户级线程

由运行在用户空间(User Space)的线程包管理、控制的线程，称为用户级线程，在这种情况下，内核感觉不到用户线程的存在，内核管理的仍然是进程。

用户级线程的最大优点是同一进程的线程之间的处理器切换不必进入内核，只需在用户态进行，极大地减少了处理器切换所带来的开销。但是，线程在运行中，如果因系统调用而阻塞，则线程所在的进程整个地被阻塞，同一进程的其他线程也不能运行。因此，从一定

意义上看，用户级线程影响了并行程度的提高。如图 3-15(a)所示，如果进程 P1 的一个线程 $T_2$ 因为 I/O 操作而进入内核而阻塞，则 P1 进程进入阻塞状态，即使线程 $T_1$ 和 $T_2$ 是就绪状态，它们也不能运行，处理器只能执行另一个进程，如 P2。




(a) 用户级线程　　(b) 系统级线程

图 3-15　线程的类型

### 2. 系统级线程

由运行在系统空间(System Space)的线程包管理、控制的线程，称为系统级线程。

与用户级线程相比，系统级线程的优缺点正好相反，因为内核具有线程的概念，在一个线程阻塞后，处理器可以选择同一进程的另一个就绪线程运行，极大地提高了并行程度；因为线程之间的切换发生在内核，比用户级线程之间的切换开销大，但也远小于进程之间的切换开销。如图 3-15(b)所示，当进程 P1 的线程 $T_2$ 因为 I/O 操作阻塞时，系统可以选择进程 P1 的线程 $T_3$ 运行，也可以选择另一个进程 P2 的一个线程，例如进程 P2 的线程 $T_2$ 运行。

## 3.8.4　线程的常用细化方法

进程细化为线程的工作是由系统设计员或程序员实现的，下面介绍三种典型的细化方法。

### 1. 分派/处理模型

根据进程的任务，把进程细化为若干个线程，其中一个线程作为协调者，称为分派线程(Dispatcher)，其他的线程作为工作者，称为处理线程(Worker)，处理线程实现具体任务的处理；分派线程根据进程当前的状态，决定处理线程的运行。这种细化方法称为分派/处理模型(Dispatcher/Worker Model)。

例如，在服务器进程中，分派线程接收一个消息时，根据消息的请求类型，选择一个处理线程，把消息提交给处理线程进一步处理，分派线程则很快地接收下一个请求消息，从而提高服务器进程的性能。如图 3-16 所示，$T_0$ 是分派线程，$T_1$、$T_2$、…、$T_n$ 为处理线程。

在分派/处理结构中，线程之间的关系具有主从关系，分派线程处于主动地位，而处理线程处于被动地位，由分派线程决定处理线程的运行。

### 2. 队列模型

一个进程，如果要完成几个独立的任务，那么，可以把进程细化为几个具有独立关系的

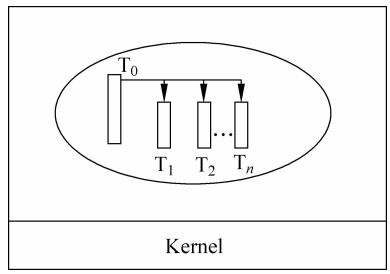



图 3-16　分派/处理模型

线程，每个线程可以单独地接收请求、处理请求和结果返回。这种的细化方法称为队列模型(Team Model)。如图 3-17 所示，一个进程细化为几个线程 $T_0$、$T_1$、…、$T_n$，它们之间独立地运行，彼此之间没有运行顺序的依赖。

在队列模型中，由于进程内部的线程相互独立，不需要同步控制，所以，队列模型的细化方法可以提高进程的运行效率。

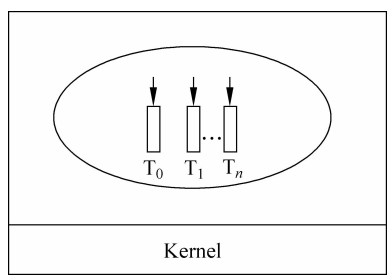



图 3-17　队列模型

### 3. 管道模型

在队列模型中，同一进程的线程之间任务相互独立。可是有的进程，其任务是对一组数据的逐步加工、处理，这样可以把进程细化为若干个线程，这些线程之间，按指定顺序依次执行，这种细化方法称为管道模型(Pipeline Model)。

如图 3-18 所示，一个进程细化为几个线程 $T_0$、$T_1$、…、$T_n$，对于一次的任务处理，这些线程只能依次地顺序运行。因此，线程之间需要同步控制。

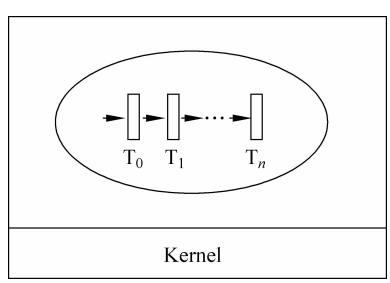



图 3-18　管道模型

上述三种模型，队列模型和管道模型是两种极端，在队列模型中，线程之间完全独立，而在管道模型中，相邻两个线程之间单向依赖，分派/处理模型则是一种折中的细化方法，其中的线程具有主从关系。

在实际的系统设计和程序实现中，进程要完成的任务多种多样，因此，系统设计员或程序员应根据应用要求，对进程进行有效的细化。

### *3.8.5 Java 线程及控制实例

Java 语言是目前应用最广泛的语言之一，正是由于 Java 基于解释器的结构风格，使得它成为主要的网络编程语言之一。这里结合一个实例，简要介绍 Java 线程的使用和基于管程的并发控制。

线程的实现和同一进程的多个线程的并发控制，都是由程序员编程完成的。在 Java 中，线程的实现有两种方式：扩展 java.lang.Thread 类和实现 java.lang.Runnable 接口。

Java 的 Lock 机制为程序员控制多个线程的同步和互斥提供了一个方便的方法。对于需要并发控制的一个程序段，可以定义 Lock 对象的一个锁，一个线程只有得到锁后才能执行对应的程序段，如果加锁时未能得到锁，则进入阻塞状态，拥有锁的线程在执行解锁操作后，可以唤醒因加锁而阻塞的线程运行。利用 Lock 机制可以实现多线程互斥访问一个管程。

Lock 机制可以用于实现多个线程的互斥关系，再结合 Java 的条件变量(Condition)机制，还可以实现多个线程的同步关系。

在 Java 的条件变量(Condition)机制中，条件变量就是表示条件的一种变量，这里的条件可以根据程序中的具体应用，赋予其含义，每个条件变量对应一个等待队列。当条件不满足时，通过调用 await()方法，线程进入阻塞状态，并加入条件变量对应的等待队列。条件变量的等待队列中的线程，由 signal()或 signalAll()方法唤醒，分别唤醒条件变量等待队列中的一个或全部线程。

条件变量实现 java.util.concurrent.locks.Condition 接口，通过一个 Lock 对象上调用 newCondition()方法进行实例化，因此，Java 中的条件变量只能和锁配合使用，但是，一个锁可以对应多个条件变量。

下面通过一个例子，介绍 Java 应用 Lock 机制和条件变量(Condition)机制的管程实现，进行线程的同步、互斥控制。

假定在一个库存管理系统中，Monitor_Stock 为库存对象，提供入库操作 importing()和出库操作 exporting()，在多个线程共享 Monitor_Stock 对象时，importing()和 exporting()需要互斥执行。另外，当库存量 stockNum 为 0 时不能执行出库操作 exporting()，并假设库存量 stockNum 不得超过 maxStockNum，使得 importing()和 exporting()之间又具有同步关系。

源程序代码如下。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;

/** 定义一个库存对象的管程 */
class Monitor_Stock {
    private int stockNum;                                    //当前库存量
    private String leastUser;                                //最近操作人
    private int maxStockNum = 3000;                          //限定最大库存量
    private Lock lock = new ReentrantLock();                 //锁变量
    private Condition _export = lock.newCondition();         //出库操作条件变量
    private Condition _import = lock.newCondition();         //入库操作条件变量

    public String getLeastUser() {
       return leastUser;
    }
    Monitor_Stock(String userName, int stockNum) {
       this.stockNum = stockNum;
       this.leastUser = userName;
    }
    public int getstockNum(){
       return this.stockNum;
    }
    /** 入库操作 */
    public void importing(int x, String userName) {
       lock.lock();                                          //加锁
       try {
           while (stockNum + x > maxStockNum) {
                System.out.println("阻塞:" + userName + "入库 = " + x + ",现库存量 = " +
stockNum);
           _import.await();                                  //阻塞入库操作
           }
           stockNum += x;                                    //入库
           leastUser = userName;
           System.out.println(userName + "本次入库 = " + x + ",现库存量 = " + stockNum);
           _export.signalAll();                              //唤醒等待该条件变量所有的线程.
       } catch (InterruptedException e) {
           e.printStackTrace();
       } finally {
           lock.unlock();                                    //解锁
       }
    }

    /** 出库操作 */
    public void exporting(int x, String userName) {
       lock.lock();                                          //加锁
       try {
           while (stockNum - x < 0) {
                System.out.println("阻塞:" + userName + "出库 = " + x + ",现库存量 = " +
stockNum);
               _export.await();                              //阻塞出库操作
           }
           stockNum -= x;                                    //出库
```

```
            leastUser = userName;
            System.out.println(userName + "本次出库 = " + x + ",现库存量 = " + stockNum);
            _import.signalAll();                          //唤醒等待该条件变量的所有入库操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
        lock.unlock();                                    //解锁
        }
    }
}
/** 入库线程类 */
class ImportThread implements Runnable {
    private String userName;                              //操作人
    private int x;                                        //本次入库数量
    Monitor_Stock Stock;

    ImportThread(String userName, int x,Monitor_Stock Stock) {
       this.userName = userName;
       this.x = x;
       this.Stock = Stock;
    }
    public void run() {
       Stock.importing(x, userName);
    }
}
/** 出库线程类 */
class ExportThread implements Runnable {
   private String userName;                               //操作人
   private int x;                                         //本次出库数量
   Monitor_Stock Stock;

   ExportThread(String userName, int x,Monitor_Stock Stock) {
      this.userName = userName;
      this.x = x;
      this.Stock = Stock;
   }
   public void run() {
      Stock.exporting(x, userName);
   }
}
/** 主程序 */
public class StockManager {
   public static void main(String[] args) {
      //创建一个共享的某产品库存对象的管程
      Monitor_Stock Stock = new Monitor_Stock("John", 0);
      //创建一个线程池,其中的线程并发执行
      ExecutorService threadPool = Executors.newFixedThreadPool(8);

      Thread t1 = new Thread(new ImportThread("Mary",2000,Stock));
      Thread t2 = new Thread(new ImportThread("Jack",2000,Stock));
```

```
        Thread t3 = new Thread(new ImportThread("Simth",2000,Stock));
        Thread t4 = new Thread(new ExportThread("Dann",2000,Stock));
        Thread t5 = new Thread(new ExportThread("Lola",2300,Stock));
        Thread t6 = new Thread(new ExportThread("Nanci",800,Stock));
        Thread t7 = new Thread(new ExportThread("OkiSan",200,Stock));
        Thread t8 = new Thread(new ExportThread("Franki",700,Stock));

        threadPool.execute(t1);
        threadPool.execute(t2);
        threadPool.execute(t5);
        threadPool.execute(t6);
        threadPool.execute(t7);
        threadPool.execute(t8);
        threadPool.execute(t4);
        threadPool.execute(t3);

        //关闭线程池
        threadPool.shutdown();
        while(!threadPool.isTerminated());
        System.out.println("最终库存量 = " + Stock.getstockNum() + ",操作人 = " +
Stock.getLeastUser());
        }
    }
```

运行结果如下。

```
Mary 本次入库 = 2000,现库存量 = 2000
阻塞:Jack 入库 = 2000,现库存量 = 2000
阻塞:Lola 出库 = 2300,现库存量 = 2000
Nanci 本次出库 = 800,现库存量 = 1200
阻塞:Jack 入库 = 2000,现库存量 = 1200
OkiSan 本次出库 = 200,现库存量 = 1000
Jack 本次入库 = 2000,现库存量 = 3000
Lola 本次出库 = 2300,现库存量 = 700
Franki 本次出库 = 700,现库存量 = 0
阻塞:Dann 出库 = 2000,现库存量 = 0
Simth 本次入库 = 2000,现库存量 = 2000
Dann 本次出库 = 2000,现库存量 = 0
最终库存量 = 0,操作人 = Dann
```


处理器是计算机系统最主要的资源,操作系统提出处理器的并发执行工作方式,实现多道程序设计,充分发挥了系统资源的利用率。进程是操作系统最重要、最基本的概念,操作系统对处理器的管理转化为对进程的管理,实现进程的并发执行。

本章首先介绍并发执行方式及其复杂性，由此引入进程的概念，从原理上介绍进程的特征、动态性，特别是进程基本状态及其转换关系。然后介绍在操作系统软件中进程的表示，即进程控制块的结构设计。

进程管理的主要功能是进程的控制、同步、通信、调度和死锁。进程控制是对进程状态转换的实现，通过一组原语实现进程控制，主要介绍进程创建原语的主要操作和进程创建时机，进程阻塞和唤醒原语的主要操作。

进程同步是对进程并发执行的协调，通过进程同步机制保证程序的可再现性，同步机制是操作系统的重点内容之一。并发进程的制约关系分为间接制约和直接制约，分别对应进程互斥关系和同步关系，同步机制是实现互斥关系和同步关系的方法。在互斥关系中，主要包括临界资源、临界区含义，加锁机制（Lock）及其分析，临界区管理准则用于验证同步机制实现互斥关系的有效性。信号量（Semphore）机制是最经典、应用最广泛的一种同步机制，主要包括信号量机制的原理、实现互斥关系的模型、实现同步关系的模型，生产者/消费者问题、读者/写者问题的并发程序设计及分析。另外，还介绍标志位机制和管程（Monitor）机制，学习、理解这两类同步机制的并发程序设计思想可以提高编程能力。

进程通信是实现进程之间任务协作的前提，消息缓冲通信是最基本的通信方式，消息缓冲通信方式的思想、设计和实现是进程通信的基础。

本章最后介绍了系统工作单位粒度的细化，即线程技术，在具有线程的操作系统中，线程是处理器调度的基本单位，进程是资源分配的基本单位。主要包括线程的含义、引入的目的。线程类型分为用户级线程和系统级线程，用户级线程极大地减少了处理器切换所带来的开销，系统级线程极大地提高了并行程度，进程细化方法主要是分派/处理结构、队列结构和管道结构。

### 1. 知识点

（1）系统工作方式及特点。
（2）进程定义。
（3）进程的特征、基本状态。
（4）PCB 及其作用。
（5）临界资源、临界区。
（6）进程控制原语。
（7）互斥、同步关系。
（8）信号量机制定义。
（9）进程通信含义。
（10）线程、线程分类，线程引入目的。

### 2. 原理和设计方法

（1）进程基本状态转换关系。
（2）信号量机制的并发程序设计模型。
（3）生产者/消费者问题及并发程序设计。
（4）读者/写者问题及并发程序设计。

(5) Dekker 算法和 Peterson 算法。

(6) 消息缓冲通信的设计与实现。

(7) 标志位机制的算法有效性分析。

(8) 霍尔管程的思想和应用。

(9) 进程与线程的关系。

## 习题

(1) 两道系统程序 A、B,共享一个整型变量 count,其代码如下。

```
A(){
    count = count + 1;
    printf("count = %d",count);
}
B(){
    count = 0;
    count = count + 10;
}
```

假定 count 初值为 100,那么,在多道程序设计环境下,A、B 各执行一次,请给出 printf()所有可能的输出结果。

(2) 请简述并发执行的思想。

(3) 什么是进程? 进程有哪些特征?

(4) 进程的三个基本状态是什么? 请画出它们之间的转换关系图。

(5) 简述进程创建的时机及进程创建原语的主要操作。

(6) 什么是临界资源? 什么是临界区?

(7) 如果把临界区设计为原语,那么,也可以实现互斥。这种做法有什么不足?

(8) 简述进程的同步关系。

(9) 在信号量机制中,p 操作的作用是什么?

(10) [条件消费 PC 问题]假定有三个进程 R、W1、W2 共享一个变量 B,进程 R 每次从输入设备上读一个整数并存入 B 中; 若 B 是奇数,则允许进程 W1 将其取出打印; 若 B 是偶数,则允许进程 W2 将其取出打印。进程 R 必须在 W1 或 W2 取出 B 中的数后才能存入下一个数,请用信号量机制实现 R、W1 和 W2 三个进程的并发执行。

(11) [重复消费 PC 问题]有三个进程 R、D、S 共享一个缓冲区 buf,进程 R 每次从输入设备上读一组数据并存入缓冲区 buf 中; 进程 D 把缓冲区中的数据取出并在屏幕上显示,同时,进程 S 把缓冲区 buf 中的数据取出并保存到磁盘上。规定: R 每次读取的数据,都要在屏幕上显示,同时也要保存到磁盘上,请用信号量机制实现 R、D、S 三个进程的并发执行。

(12) 今有三个并发进程 R、M、P,如图 3-19 所示,它们的任务如下:

R 负责从输入设备读取记录信息,每读一个记录后,把它存放在缓冲区队列 Buf1。

M 从缓冲区队列 Buf1 中读取记录,读出后加工记录并把结果存入缓冲区队列 Buf2 中。

P 从缓冲区队列 Buf2 中读取记录打印输出。

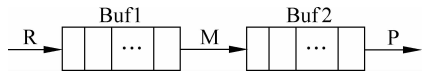



图 3-19　习题(12)

假定缓冲区队列 Buf1 有 $n$ 个缓冲区，缓冲区队列 Buf2 有 $m$ 个缓冲区，$n>1$，$m>1$，一个记录存放在一个缓冲区。请用信号量机制实现 R、M 和 P 的并发执行。

(13) 有三个进程 P1、P2 和 C，它们共享一个缓冲区 buf。进程 P1 反复地从设备上读一个记录信息，并将其存入缓冲区 buf；进程 P2 反复地从另一个设备上读一个记录信息，也将其存入缓冲区 buf；进程 C 将缓冲区 buf 中的记录信息取出，并加工处理。如果缓冲区 buf 只能存储一个记录，只有在进程 C 读取信息后，才能存储下一个记录，同时规定，P1 或 P2 不能连续两次向缓冲区 buf 存放记录，且在初始状态它们中哪一个先向缓冲区 buf 存放信息都是允许的。请用信号量机制实现进程 P1、P2 和 C 的并发执行。

(14) 在读者/写者问题中，假定至多只有 $N$ 个($N>1$)读者进程同时执行读者操作。请给出并发程序设计。

(15) 已知两个互斥关系的进程 $P_0$、$P_1$ 并发程序设计如下。

```
boolean flag[2] = {false,false};
P0()
{
    …
    flag[0] = true;
      while(flag[1] == true){
        flag[0] = false;
        flag [0] = true;
    }
    临界区;
    flag[0] = false;
    …
}
P1()
{
    …
    flag[1] = true;
    while(flag[0] == true){
        flag[1] = false;
        flag [1] = true;
    }
    临界区;
    flag[1] = false;
    …
}
```

试分析这里的设计是否满足临界区管理准则(1)和准则(2)?

(16) 给出 $n$ 个进程互斥($n>2$)的 Peterson 算法的并发程序设计。

(17) 操作系统为什么要提供进程通信?

(18) 请从同步关系的角度,描述消息缓冲通信的实现过程。

(19) 解释引入线程的主要目的。

(20) 进程的一个线程与该进程的一个子进程有何区别?

(21) 线程有哪两种基本类型?各有哪些主要的优缺点?