第 3 章　进程和线程

3.1　进程的概念

在单道批处理系统中，计算机一次只能执行一个程序，该程序完全控制计算机系统的运行。在这样的计算机系统中，所有的系统资源都为该程序使用。这种方式存在资源浪费、系统运行效率低等问题。为了提高资源利用率和系统的吞吐量，现代计算机系统采用多道程序技术，允许同时加载多个程序到内存中，并使之并发执行。在多道程序环境下，由于处理器需要在各程序模块之间来回切换，程序的执行具有间断性。此外，由于并发执行的程序共享系统中的资源，任一程序对这些资源状态的改变都会影响其他程序的运行环境，造成执行中的程序模块之间存在制约关系，甚至改变程序的执行结果。怎样描述程序在执行过程中对计算机资源的需求？这些资源的变化对程序的执行结果有哪些影响？显然，为了解答上问题，只有程序的概念是不够的。程序只是对计算任务和数据的静态描述，无法刻画并发执行过程带来的新特征。为了描述和控制资源共享和竞争对并发执行的程序和执行结果的影响，本章引入进程和线程概念。

3.1.1　程序的顺序执行和并发执行

1. 程序

程序是描述计算机要完成的独立功能，并在时间上按严格次序前后相继的计算机操作序列的集合，是一个静态的概念。它体现了编程人员确定的计算机完成相应功能时应该采取的步骤。

2. 程序的顺序执行

程序只有经过执行才能得到结果。程序的执行可以分为顺序执行和并发执行。计算机 CPU 是通过时序脉冲来控制顺序执行指令的。其执行过程可以描述为

```
repeat IR←M[PC]
  PC←PC+1
  execute (instruction in IR)
until CPU halt
```

这里 IR 为指令寄存器，PC 为程序计数器，M 为存储器。显然，程序的顺序性与计算机硬件的顺序性是一致的。一个具有独立功能的程序独占处理器直至最终结束的过程称为程序的顺序执行。程序的顺序执行具有如下特点：

(1) 顺序性。程序顺序执行时，其执行过程可看作一系列严格按程序规定进行状态转移的过程，即，每执行一条指令，系统将从上一个执行状态转移到下一个执行状态，且上一条

指令的执行结束是下一条指令执行开始的充分必要条件。

(2) 封闭性。程序执行得到的最终结果由给定的初始条件决定,不受外界因素的影响。

(3) 可再现性。程序顺序执行的最终结果与执行速度和环境无关。只要输入的初始条件相同,则无论何时重复执行该程序,都会得到相同的结果。

3. 多道程序系统中程序执行环境的变化

如果计算机在任一时刻都只处理一个具有独立功能的程序,操作系统的设计和功能都将变得非常简单,因为在这样的系统中不存在资源共享、程序的并发执行以及用户执行的随机性问题。但是,在许多情况下,计算机需要能够同时处理多个具有独立功能的程序。批处理系统、分时系统、实时系统、网络系统和分布式系统等都是这样的系统。因此,每个程序在执行时都应考虑其执行环境的变化。多道程序系统中的执行环境具有下述 3 个特点:

(1) 独立性。在多道程序系统中执行的每道程序都是逻辑上独立的,它们之间不存在逻辑上的制约关系。也就是说,如果有充分的资源保证,则每道程序都可以独立执行,且执行速度与其他程序无关,执行的起止时间也是独立的。

(2) 随机性。在多道程序环境下,特别是在多用户环境下,程序和数据的输入与执行开始时间都是随机的。在实时系统中更是如此,它要在规定的短时间内对随机的输入做出反应。输入与程序执行开始时间的随机性向操作系统提出了同时处理多道程序的客观要求。

(3) 资源的共享与竞争。这里所说的资源,既包括硬件资源,也包括软件资源。硬件资源包括 CPU、输入输出设备、存储器等。软件资源除了各种程序之外,还包括各种可共享的数据。显然,任何一个计算机系统中的软硬件资源都是有限的,特别是在单 CPU 系统中。一般来说,多道程序环境下执行程序的道数超过计算机系统中 CPU 的个数,单 CPU 系统更是如此。显然,受 CPU 个数的限制,由随机性引起的需同时执行的 N 道($N \geqslant 2$) 程序只能共享系统中已有的 CPU。在单 CPU 系统中,则有 $N-1$ 道程序处于等待 CPU 的状态。同理,输入输出设备、内存等都存在共享与竞争。资源的共享与竞争会制约进程的执行速度,改变其执行环境与执行结果。

4. 程序的并发执行

1) 什么是程序的并发执行

所谓并发执行,是为了增强计算机系统的处理能力和提高资源利用率所采取的一种同时执行技术。程序的并发执行可进一步分为两种。

第一种并发执行是多道程序在执行时由于资源竞争而造成的并发执行。如前所述,在多道程序环境下,每道程序在逻辑上独立,具备了独立执行的条件。而程序与数据输入的随机性以及执行起始时间的随机性又导致了多道程序同时要求计算机系统的资源。由于资源有限,多道程序的同时执行要求必然造成资源的共享与竞争,从而,在一些程序执行时,另外一些程序会等待或抢占正在执行的程序的资源。因此,这种同时执行无法作到在微观上(也就是在指令级上)的同时执行。

第二种并发执行是在某道程序的几个程序段中包含一部分可以同时执行或颠倒顺序执行的代码。例如:

```
read(a);
```

```
read(b);
```

它们既可以同时执行，也可颠倒顺序执行。也就是说，对于这样的语句，同时执行不会改变顺序程序具有的逻辑性质。因此，可以采用并发执行来充分利用系统资源，以提高计算机的处理能力。

综上所述，程序的并发执行可总结为：一组在逻辑上互相独立的程序或程序段在执行过程中，其执行时间在宏观上互相重叠，而在微观上互相竞争、互相等待，即一个程序段的执行尚未结束，另一个程序段的执行已经开始的执行方式。

程序的并发执行不同于程序的并行执行。程序的并行执行是指一组程序按独立的、异步的速度执行。并行执行不等于时间上的重叠。可以将并发执行过程描述为

```
S0
cobegin
    P1; P2; …; Pn
coend
Sn
```

这里，S_0、S_n 分别表示并发程序段 P_1，P_2，…，P_n 开始执行前和并发执行结束后的语句。即，先执行 S_0，再并发执行 P_1，P_2，…，P_n；P_1，P_2，…，P_n 全部执行完毕后，再执行 S_n。P_1，P_2，…，P_n 也可以由同一程序段中的不同语句组成。1966 年，Bernstein 提出了两个相邻语句 S_1、S_2 可以并发执行的条件：

将程序中任一语句 S_i 划分为两个变量的集合 $R(S_i)$ 和 $W(S_i)$。其中

$$R(S_i)=\{a_1,a_2,\cdots,a_m\},\quad a_j(j=1,2,\cdots,m)$$

是语句 S_i 在执行期间必须对其进行读写的变量；而

$$W(S_i)=\{b_1,b_2,\cdots,b_n\},\quad b_j(j=1,2,\cdots,n)$$

是语句 S_i 在执行期间必须对其进行修改、访问的变量。

如果对于语句 S_1 和 S_2，有

$$R(S_1)\cap W(S_2)=\varnothing$$
$$W(S_1)\cap R(S_2)=\varnothing$$
$$W(S_1)\cap W(S_2)=\varnothing$$

同时成立，则语句 S_1 和 S_2 是可以并发执行的。

2）程序的并发执行带来的影响

程序的并发执行可以充分地利用系统资源，从而提高系统的处理能力，这是并发执行好的一方面。但是，正如前面提到的那样，由于系统资源有限，程序的并发执行必然导致资源共享和资源竞争，从而改变程序的执行环境和速度。那么，由资源共享和竞争引起的程序执行速度的改变是否会对程序的最终结果带来不利的影响呢？也就是说，是否会保持用户期望的执行结果的封闭性和再现性呢？如果并发执行的各程序段中的语句或指令满足上述 3 个条件，则认为并发执行不会对执行结果的封闭性和再现性产生影响（证明略）。但是，在一般情况下，系统要判定并发执行的各程序段是否满足上述 3 个条件是相当困难的。从而，如果并发执行的程序段不按照特定的规则和方法进行资源共享和竞争，则其执行结果将不可避免地失去封闭性和再现性。下面的例子说明了这一点。

例如，设有堆栈 S，栈指针为 top，堆栈中存放内存中相应数据块的地址，如图 3.1(a)所示。设有两个程序模块 getaddr(top)和 reladdr(blk)，其中，getaddr(top)从给定的 top 所指的堆栈中取出相应的内存数据块地址，而 reladdr(blk)则将内存数据块地址 blk 放入堆栈 S 中。getaddr(top)和 reladdr(blk)可分别描述为

```
procedure getaddr(top)
  begin
    local r
    r←(top)
    top←top-1
    return(r)
  end
procedure reladdr(blk)
  begin
    top←top+ 1
    (top)←blk
  end
```

显然，如果 getaddr 和 reladdr 这两个程序段采用顺序执行，其执行结果具有封闭性和再现性。但是，如果这两个程序段采用并发执行，则在单处理器系统中，将有可能出现下述情况。

首先，程序段 reladdr 开始执行，准备将内存数据块地址放入堆栈。然而，当 reladdr 执行到 top←top＋1 语句时(如图 3.1(b))，程序段 getaddr 也开始执行且抢占了处理器，从而使程序段 reladdr 停在 top←top＋1 处等待处理器，如图 3.1(b)所示。getaddr 程序段的执行目的是从堆栈指针 top 所指的堆栈格中取出一个内存数据块地址，显然，由于 reladdr 程序段的执行将指针 top 升高了一格且未放进适当的数据，getaddr 的执行结果是取数失败，如图 3.1(c)所示。另外，如果改变程序段 getaddr 和 reladdr 的执行顺序或执行速度，又可得到不同的执行结果。这说明了如下问题：在某些情况下，程序的并发执行使得其执行结果不再具有封闭性和再现性，且可能造成程序出现错误。

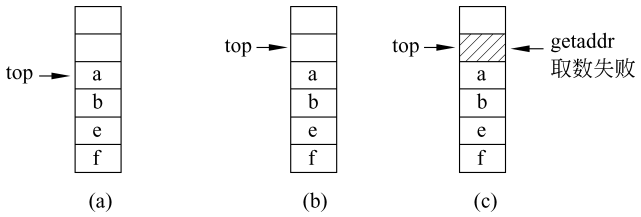

图 3.1　堆栈的取数和存数过程

上例中的程序段并发执行出现错误结果是由于两个程序段共享堆栈，从而使得执行结果受执行速度影响。一般情况下，并发执行的各程序段如果共享软硬件资源，都会造成其执行结果受执行速度影响的局面。显然，这是程序设计人员不希望看到的。为了使得在并发执行时不出现错误结果，必须采取某些措施来制约、控制各并发程序段的执行速度。这在操作系统程序设计中尤其重要。为了控制和协调各程序段执行过程中的软硬件资源的共享和

竞争，显然，必须有一个描述各程序段在执行过程中需求和使用资源情况的基本单位。从上述讨论可以看出，由于程序的顺序性、静态性以及孤立性，用程序段无法描述其在执行过程中是否得到了它需要的资源。需要有一个能描述程序的执行过程且能描述在执行过程中在何时何地使用了何种资源的基本单位。这个基本单位被称为进程。

事实上，上述对处理器的竞争造成执行结果发生错误的现象还可以进一步延伸到用管道指令执行的系统。一条指令在一条管道上执行时，如果恰逢另一条指令访问该管道上的数据，则可能产生执行错误。

3.1.2 进程的定义

与作业相比，进程是一个动态概念。它是20世纪60年代初期首先在Multics系统和IBM公司的TSS/360系统中提出的，其目的是把程序的静态状态和动态的执行过程区别开来。从那以来，人们对进程下过许多各式各样的定义。现列举其中几种：

(1) 进程是可以并发执行的计算部分(S.E.Madnick，J.T.Donovan)。

(2) 进程是一个独立的可以调度的活动(E.Cohen，D.Jofferson)。

(3) 进程是一个抽象实体，当它执行某个任务时，将分配和释放各种资源(P.Denning)。

(4) 行为的规则叫程序，程序在处理器上执行时的活动称为进程(E.W.Dijkstra)。

(5) 一个进程是一系列逐一执行的操作，而操作的确切含义则有赖于以何种详尽程度来描述进程(B.Hansen)。

以上关于进程的定义尽管各有侧重，但在本质上是相同的。即注重进程是一个动态的执行过程这一点。也可以这样定义进程：进程是并发执行的程序在执行过程中分配和管理资源的基本单位。

进程和程序是两个既有联系又有区别的概念，简述如下：

(1) 程序是一个静态概念，而进程是一个动态概念。程序是指令的有序集合，没有任何执行的含义。进程则强调执行过程，它动态地被创建，并在被调度执行后消亡。如果把程序比作菜谱，那么进程则是按照菜谱炒菜的过程。

(2) 进程具有并发特征，而程序没有。从进程的定义可知，进程具有并发特征的两方面，即独立性和异步性。也就是说，在不考虑资源共享的情况下，进程的执行是独立的，执行速度是异步的。显然，由于程序不反映执行过程，所以不具有并发特征。

(3) 进程是共享计算机系统资源的基本单位，从而其并发性受到系统的制约。这里，制约是指限制进程的独立性和异步性。

(4) 不同的进程可以包含同一程序，只要该程序对应的数据集不同。

3.2 进程的描述

3.2.1 进程控制块

进程是一个动态概念。系统需要用一个数据结构来描述和管理它，这个数据结构称为进程控制块(PCB)。PCB包含3部分信息：进程的描述信息、进程的控制信息及进程使用的资源信息。有些系统中的PCB还包含进程在调度等待时使用的现场保护区信息。PCB

集中反映一个进程的动态特征。在创建一个进程时，应首先创建其 PCB，然后才能根据 PCB 中的信息有效地管理和控制该进程。当一个进程完成其功能之后，或者需要删除一个进程时，系统会释放或删除 PCB，进程也随之消亡。

一般来说，在不同的操作系统中，PCB 包含的内容会多少有所不同。但是，下面所示的基本内容是必需的：

(1) 描述信息。主要包括下列 3 种：

① 进程名或进程标识号。每个进程都有唯一的进程名或进程标识号。在识别一个进程时，进程名或进程标识号代表该进程。

② 用户名或用户标识号。每个进程都隶属于某个用户，用户名或用户标识号有利于资源共享与保护。

③ 家族关系。描述进程之间的关系。在有的系统中，进程之间构成家族关系。因此，在 PCB 中相应的项描述其家族关系。

(2) 控制信息。主要包括下列 5 种：

① 进程当前状态。说明进程当前处于何种状态。进程在活动期间可分为初始状态、就绪状态、执行状态、等待状态和终止状态。任一进程在任一时刻只能处于这 5 种状态之一。这 5 种状态的含义如下：初始状态表示进程刚刚被创建，还处于只有数据结构的状态；就绪状态表示该进程正准备占有处理器；执行状态表示该进程已经占有处理器，正在执行；等待状态则表示进程因某种原因而暂时不能占有处理器，处于等待占有处理器的过程中；终止状态的进程则等待释放进程占有的所有资源，并等待系统将自己删除。在有的系统中，等待状态会进一步划分为不同原因或不同地点(内存或外存)的等待。有关进程的状态将在 3.3.3 节中进一步讨论。

② 进程优先级。是选取进程占有处理机的重要依据。与进程优先级有关的 PCB 表项有以下几个：

- 占有 CPU 时间。
- 进程优先级偏移。
- 占有内存时间等。

③ 程序开始地址。规定该进程的程序从此地址开始执行。

④ 各种计时信息。给出进程占有和利用资源的有关情况。

⑤ 通信信息。说明该进程在执行过程中与其他进程之间的信息交换情况。

(3) 资源管理信息。PCB 中包含得最多的是资源管理信息，包括有关存储器的信息、使用输入输出设备的信息、有关文件系统的信息等。这些信息具体如下：

① 占用内存大小及其管理用数据结构指针，例如 5.3.2 节介绍的进程页表指针等。

② 在某些复杂系统中，还有关于内存中的程序或数据覆盖与交换的有关信息，如交换程序模块长度、交换区地址等。这些信息在进程申请和释放内存时使用。有关程序的覆盖与交换，将在 5.5 节中讲述。

③ 共享程序段长度及起始地址。

④ 输入输出设备的设备号、要传送的数据长度、缓冲区地址、缓冲区长度及所用设备的有关数据结构指针等。这些信息在进程申请和释放设备以进行数据传输时使用。

⑤ 指向文件系统的指针及有关标识等。进程可使用这些信息对文件系统进行操作。

(4) CPU 现场保护区。当前进程因等待某个事件而进入等待状态或因某种事件发生而中止 CPU 的执行时,为了以后该进程能回到被中止处恢复执行,就需要保护当前进程的 CPU 执行现场。PCB 中设有专门的 CPU 现场保护区,以存储 CPU 中止执行时的进程现场。

PCB 是操作系统感知进程存在的唯一实体。通过对 PCB 的控制和管理,操作系统为有关进程分配资源,调度有关进程执行。当进程执行结束时,则通过删除 PCB 来释放进程占有的各种资源。

由于 PCB 中包含较多的信息,因此,一个 PCB 往往要占据较大的存储空间(一般占几百到几千字节)。在有的系统中,为了减少 PCB 对内存的占用量,只允许 PCB 中最常用的部分,如 CPU 现场保护信息、进程描述信息、进程控制信息等常驻内存;PCB 结构中的其他部分则存放于外存之中,待该进程将要执行时才与其他数据一起装入内存。

3.2.2 进程上下文及其切换

进程是一个动态概念,但它也有自己的静态描述。进程的静态描述由 PCB、进程使用的有关程序段和数据集组成。上面已经介绍了 PCB,本节介绍进程使用的程序段和数据集。如果把进程的执行过程从头至尾地记录下来,在每条指令的前后就形成了静态的上下文关系。

进程上下文实际上是供进程切换用的。它是一个与进程调度和处理器状态变化有关的概念。在进程执行过程中,由于中断、出错等原因造成进程被悬挂,这时操作系统需要知道和记忆进程已经执行到什么地方或新的进程将从何处执行。另外,进程执行过程中还经常出现一个程序调用其他程序的情况,这时也需要保存和记忆进程的当前状态。

已执行的进程指令和数据在寄存器与堆栈中的内容称为上文,正在执行的指令和数据在寄存器与堆栈中的内容称为正文,待执行的指令和数据在寄存器与堆栈中的内容称为下文。

在不发生进程调度时,进程上下文的改变都是在同一进程内进行的。此时,一条指令的执行对进程上下文的改变较小,一般反映为指令寄存器、程序计数器以及用于保存调用子程序返回接口的堆栈值等的变化。

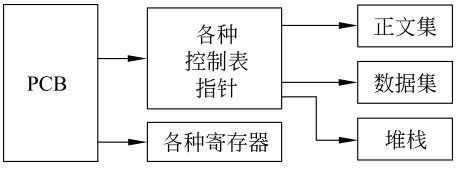

图 3.2 进程上下文结构

同一进程的上下文的结构由与执行该进程有关的各种寄存器中的值、程序段经过编译后形成的机器指令代码集(或称正文集)、数据集及各种堆栈值与 PCB 构成,如图 3.2 所示。

这里,有关寄存器和堆栈的内容是重要的。例如,没有程序计数器(PC)和程序状态寄存器(PS),CPU 将无法知道下一条待执行指令的地址,无法控制有关操作。

图 3.3 是 UNIX System Ⅴ的进程上下文组成示例。在 UNIX System Ⅴ中,进程上下文由用户级上下文、寄存器上下文以及系统级上下文组成。

进程的用户级上下文由进程的用户程序段部分编译而成的正文段、数据、栈等组成。

进程的寄存器上下文由 PC、PS、栈指针和通用寄存器的值组成。其中,PC 给出 CPU 将要执行的下一条指令的虚地址;PS 给出计算机与该进程相关联时的硬件状态,例如当前

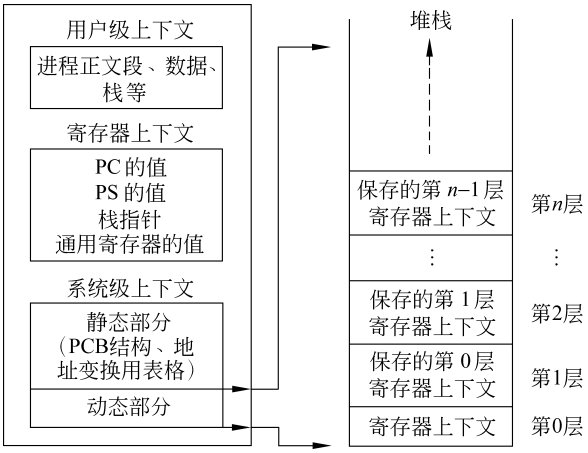

图 3.3 UNIX System Ⅴ进程上下文组成

执行模式、能否执行特权指令等；栈指针指向下一项的当前地址；通用寄存器用于不同执行模式之间的参数传递等。

进程的系统级上下文又分为静态部分与动态部分。这里的动态部分不是指程序的执行，而是指在进入和退出不同的上下文层次时，系统为各层上下文中相关联的寄存器值所保存和恢复的记录。

系统级上下文的静态部分包括 PCB 结构、用于将进程的虚地址空间映射到物理空间的有关表格和内核堆栈。这里，内核堆栈主要用来装载进程中使用的系统调用的序列。

系统级上下文的动态部分是与寄存器上下文相关联的。进程上下文的层次概念主要体现在系统级上下文的动态部分中，即系统级上下文的动态部分可看成由一些数量变化的层次组成。其变化规则满足先进后出的堆栈方式，每个上下文层次在堆栈中各占一项。

进程上下文切换发生在不同的进程之间，而不是发生在同一个进程内。

如果操作系统的内核态（操作系统内核状态）程序执行和用户态（非操作系统内核状态）程序执行在一个进程中完成，则同一进程内内核态和用户态之间的转换也会发生进程上下文切换。

进程上下文切换过程一般包含 3 部分，并涉及 3 个进程。

第一部分是保存被切换进程的正文部分（或当前状态）至有关内存，例如该进程的 PCB 中。

第二部分是操作系统进程中有关的调度和资源分配程序执行，并选取新的进程。

第三部分是将被选中进程原来被保存的正文部分从有关内存中取出，并送至有关寄存器与堆栈中，激活被选中进程执行。

进程上下文切换过程如图 3.4 所示。

进程上下文的切换过程涉及由谁来保护和获取进程的正文的问题，也就是如何使寄存器和堆栈中的数据流入、流出 PCB 的内存。

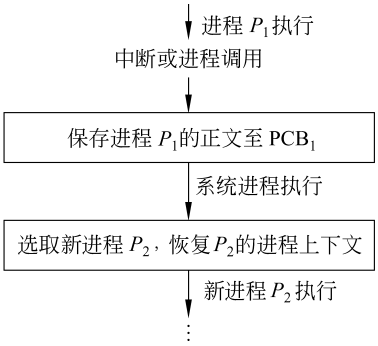

图 3.4 进程上下文切换过程

进程上下文切换还涉及系统调度和分配程序，这些都比较耗费 CPU 时间。

为了提高系统执行效率，有的计算机在设计时采用了多组寄存器技术。即，在进行进程上下文切换时，不保留被切换进程上下文的正文，但是切换进程执行时使用的寄存器。这样就减少了数据保存和获取所耗费的时间。

近年来，为了进一步提高系统执行效率，人们又提出了线程的概念。有关线程的内容将在 3.8 节中讲述。

3.2.3 进程空间和大小

任一进程都有自己的地址空间，该空间称为进程空间或虚拟地址空间(有关虚拟地址空间的概念，将在 5.4 节中讲述)。进程空间的大小只与处理器的位数有关。例如，16 位处理器的进程空间大小为 2^{16}，而 32 位处理器的进程空间大小为 2^{32}。程序的执行都在进程空间内进行。用户程序、进程的各种控制表格等都按一定的虚拟地址结构排列在进程空间中。另外，在早期的 UNIX 以及 Linux 等操作系统中，进程空间还被划分为用户空间和系统空间两大部分，如图 3.5 所示。

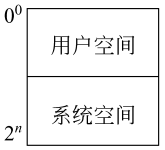

图 3.5 进程空间的划分

在进程空间被划分为两大部分后，用户程序在用户空间内执行，而操作系统内核程序则在进程的系统空间内执行。

另外，为了防止用户程序访问系统空间，造成访问出错，计算机系统还通过程序状态寄存器等设置不同的执行模式，即用户执行模式和系统执行模式来进行保护。人们也把用户执行模式和系统执行模式分别称为用户态和系统态。由于用户空间和系统空间的地址之间不能简单地连续访问，因此，当进程中的程序在用户态和系统态之间转换时，也会发生进程上下文切换。

3.3 进程控制

进程和处理器管理的一个重要任务是进程控制。所谓进程控制，就是系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态间的转换，从而达到多个进程高效率并发执行和资源共享的目的。

在系统态下执行的某些具有特定功能的程序段称为原语。原语可分为两类：一类原语是机器指令级的，其特点是执行期间不允许中断，正如物理学中的原子一样，在操作系统中，原语是一个不可分割的基本单位；另一类原语是功能级的，其特点是作为原语的程序段不允许并发执行或在并发执行时必须保证其顺序执行的结果，使其执行结果具有封闭性和再现性。

这两类原语都在系统态下执行，而且都是为了完成某个系统管理所需的功能并被高层软件所调用。

显然，系统在创建、撤销一个进程以及改变进程的状态时，都要调用相应的程序段完成这些功能。那么，这些程序段是不是原语呢？如果它们不是原语，则由上述原语的定义可知，这些程序段是允许并发执行的。然而，如果不加控制和管理地让这些控制进程状态转换及创建和撤销进程的程序段并发执行，则会使得其执行结果失去封闭性和再现性，从而达不

到进程控制的目的。反过来，如果对这些程序段采用下面几节中所述的控制方法使其在并发执行过程中也能完成进程控制任务，将会大大增加系统的开销和复杂度。因此，在操作系统中，通常用原语控制进程。用于进程控制的原语有创建原语、撤销原语、阻塞原语、唤醒原语等。

3.3.1 创建与撤销

1. 进程创建

进程创建方式有以下几种：

（1）由系统程序的原语创建。例如，在批处理系统中，由操作系统的作业调度程序（原语）为用户作业创建相应的进程，以完成用户作业所要求的功能。

（2）由父进程创建。例如，在进程家族是层次结构的系统中，父进程创建子进程以完成并行工作。

由系统原语创建的进程之间的关系是平等的，它们之间一般不存在资源继承关系。而在父进程和由其创建的子进程之间则存在隶属关系，且构成树状结构。属于某个家族的进程可以继承和使用其父进程拥有的资源。另外，无论采用哪一种方式创建进程，在操作系统被引导进入内存时，都必须创建承担系统资源分配和管理工作的系统进程。

无论是系统创建方式还是父进程创建方式，都必须调用创建原语来实现。创建原语扫描系统的 PCB 链表，在找到空的 PCB 之后，填入调用者提供的有关参数，最后形成代表进程的 PCB 结构。这些参数包括进程标识、进程优先级、进程正文段起始地址、资源清单等。其实现过程如图 3.6 所示。

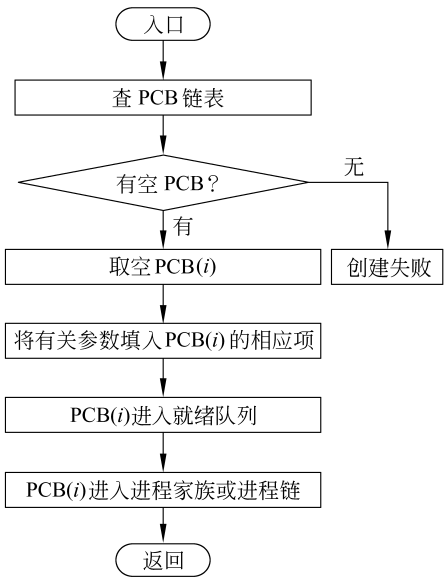

图 3.6 创建原语的实现过程

2. 进程撤销

以下几种情况导致进程被撤销：

（1）进程已完成其功能而正常终止。

（2）由于某种错误导致进程非正常终止。

（3）祖先进程要求撤销某个子孙进程。

无论哪一种情况导致进程被撤销，进程都必须释放它占有的各种资源和 PCB 结构本身，以利于资源的再利用。当然，一个进程占有的某些资源在使用结束时可能早已释放。另外，当一个祖先进程撤销某个子孙进程时，还需要审查该子孙进程是否还有自己的子孙进程。若有，还需撤销其子孙进程的 PCB 结构并释放它们占有的资源。

撤销原语首先检查 PCB 链表或进程家族，寻找要撤销的进程是否存在。如果找到了要撤销的进程的 PCB，则撤销原语释放该进程占有的资源之后，把对应的 PCB 从进程家族中摘下并将其返回给空 PCB 队列。如果被撤销的进程有自己的子进程，则撤销原语先撤销其

子进程的 PCB 并释放该子进程占有的资源之后，再撤销当前进程的 PCB 并释放其占用的资源。撤销原语的实现过程如图 3.7 所示。

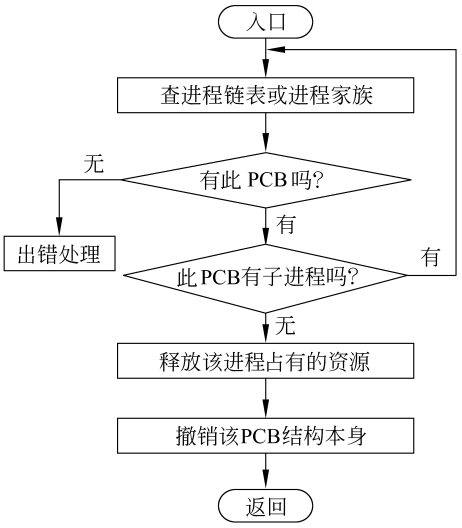

图 3.7　撤销原语的实现过程

3.3.2　阻塞与唤醒

进程的创建原语和撤销原语完成了进程从无到有、从存在到消亡的变化。被创建的进程最初处于就绪状态，经调度程序选中后进入执行状态。有关进程调度的内容将在 4.3 节中详述，这里主要介绍实现进程从执行状态到等待状态和从等待状态到就绪状态的转换的两种原语，即阻塞原语与唤醒原语。

当一个进程期待某一事件（例如键盘输入数据、写盘、其他进程发来数据等）发生，但发生条件尚不具备时，该进程会调用阻塞原语来阻塞自己。阻塞原语在阻塞一个进程时，由于该进程正处于执行状态，故应先中断处理器并保存该进程的现场。然后将被阻塞进程置为阻塞状态后送入等待队列中，再转到进程调度程序，选择新的就绪进程投入运行。阻塞原语的实现过程如图 3.8 所示。这里，转进程调度程序是很重要的，否则，处理器将会出现空转而浪费资源。

当等待队列中的进程所等待的事件发生时，等待该事件的所有进程都将被唤醒。显然，一个处于阻塞状态的进程不可能自己唤醒自己。唤醒进程有两种方法：一种是由系统进程唤醒；另一种是由事件发生进程唤醒。

当由系统进程唤醒等待进程时，系统进程要了解和掌握计算机中所有事件的发生。系统进程收到与将要唤醒的进程有关的事件后，将“事件发生”这一消息通知等待进程。从而使得该进程因等待事件已发生而进入就绪队列。

等待进程也可由事件发生进程唤醒。此时，事件发生进程和被唤醒进程之间是合作关系。因此，唤醒原语既可被系统进程调用，也可被事件发生进程调用。我们称调用唤醒原语的进程为唤醒进程。唤醒原语首先将被唤醒进程从相应的等待队列中摘下，并将被唤醒进程置为就绪状态，送入就绪队列。在把被唤醒进程送入就绪队列之后，唤醒原语既可以返回

原调用程序，也可以转向进程调度程序，以便让进程调度程序有机会选择一个合适的进程执行。唤醒原语的实现过程如图 3.9 所示。

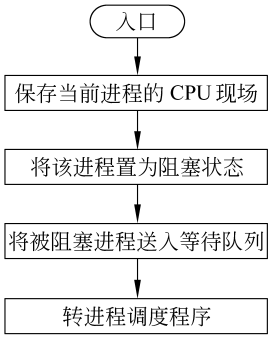

图 3.8 阻塞原语的实现过程

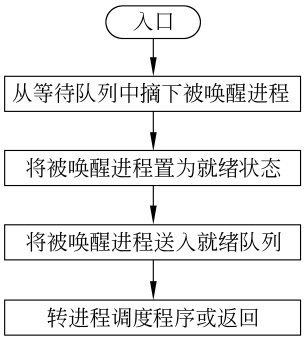

图 3.9 唤醒原语的实现过程

3.3.3 进程状态及其转换

如上所述，进程是一个动态概念。它有自己的生命周期。一个进程的生命周期可以用一组状态来描述，这些状态刻画整个进程的活动过程。进程在不同运行状态时，分别用不同的状态字描述。这些状态字存放在进程的 PCB 结构中。系统根据 PCB 结构中的状态字控制进程运行。前面已介绍过，一个进程在并发执行中，由于资源共享与竞争，有时处于执行状态，有时则因等待某种事件发生而处于等待状态。当一个处于等待状态的进程在等待事件发生后会被唤醒，又因没有立即得到处理器而进入就绪状态。进程刚被创建时，如果还未给它分配必要的资源，它只能处于初始状态。如果初始状态的进程得到了相关资源，可以抢占处理器，但由于其他进程正占有处理器而抢不到，它就会进入就绪状态。

处于就绪状态的进程由调度得到处理器，便可立即执行。

进程因等待某个事件发生而放弃处理器进入等待状态。进程在执行结束后，将退出执行而被终止，这时进程处于终止状态，释放其占有的资源。

因此，在进程的生命期内，一个进程至少具有 5 种基本状态，分别是初始状态、就绪状态、执行状态、等待状态和终止状态。

在有些系统中，为了有效地利用内存，就绪状态又可进一步分为内存就绪状态和外存就绪状态。在这样的系统中，只有处于内存就绪状态的进程在得到处理器后才能立即投入执行；而处于外存就绪状态的进程只有先进入内存就绪状态，才可能被调度执行。这种方式明显地提高了内存的利用效率，但是增加了系统开销和复杂度。

在单 CPU 系统中，处于执行状态的进程只能有一个。处于就绪状态的进程经调度选中之后才可进入执行状态。

在某些操作系统中，一个进程在其生命周期内的执行过程中，总要涉及用户程序和操作系统内核程序两部分。因此，进程的执行状态又可进一步划分为用户执行状态（简称用户态）和系统执行状态（简称系统态或内核态）。一个进程的用户程序段在执行时，该进程处于用户态；而一个进程的系统程序段在执行时，该进程处于系统态。

为什么要划分用户态和系统态呢？一个最主要的目的是要把用户程序和系统程序区分

开来，以利于系统程序的共享和保护。显然，这也是以增加系统开销和复杂度为代价的。

等待状态可根据等待事件的种类进一步划分为不同的子状态，例如内存等待状态、设备等待状态、文件等待状态和数据等待状态等。这样做的好处是系统控制简单，发现和唤醒相应的进程较为容易。但系统中设置过多的状态又会造成系统参数和状态转换过程的增加。

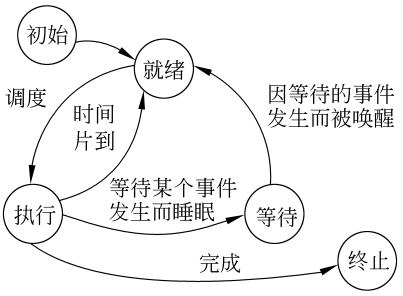

图 3.10 进程的 5 种基本状态之间的转换关系

进程的状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化发生转换。那么，是什么样的条件使得进程各状态发生转换呢？图 3.10给出了进程的 5 种基本状态，即初始状态、就绪状态、执行状态、等待状态与终止状态之间的转换关系。

事实上，进程的状态转换是一个非常复杂的过程。从一个状态到另一个状态的转换除了要使用不同的控制过程，有时还要借助于硬件触发器才能完成。例如，在 UNIX 系统中，从系统态到用户态的转换要借助硬件触发器完成。

3.4 进程互斥

3.4.1 概念

在介绍进程的概念时，已经介绍过进程具有独立性和异步性等特征。由于资源有限，进程对资源的竞争又导致其独立性和异步性受到制约。那么，在进程的并发执行过程中存在哪些制约呢？下面讨论这个问题。

1. 临界区

人们在描述一个程序或算法时，总是认为程序或算法可以按照规定的步骤执行，并且每次执行都可以得到相同的结果。事实上，在实际的系统中往往不是这样。一般来说，编写程序时使用的都是高级语言。程序中的一条语句往往是由多条执行指令构成的。例如，程序中有以下赋值语句：

```
X=X+1;
```

在用汇编语言书写时，就变成了以下 3 条语句：

```
LOAD A, X
ADDI A, 1
STORE A, X
```

这里 A 代表累加器。根据系统的设计和要求，在这 3 条语句执行期间，也有可能发生中断或调度，从而使得与当前进程无关的程序得以执行，改变累加器中的内容。为了保证程序执行最终结果的正确性，必须对并发执行的各程序段进行制约，以控制它们的执行速度和对资源的竞争。在 3.1.1 节中已经介绍了进程中两条相邻语句可以并发执行的 3 个条件。可是，

在实际系统中，要检验即将执行的两条相邻语句是否满足这 3 个条件需要巨大的系统开销。那么，是否有更简单的办法能够检查出需要对程序的哪些部分进行制约才能保证其执行结果的正确性呢？下面看一个例子。

设有两个计算进程 PA、PB 共享内存 MS，其中 MS 分为 3 个领域，即系统区、进程工作区和数据区，如图 3.11 所示。这里数据区被划分大小相等的块，每个块中既可能存放了数据，也有可能没有数据。系统区主要是堆栈，其中存放空数据块的地址。

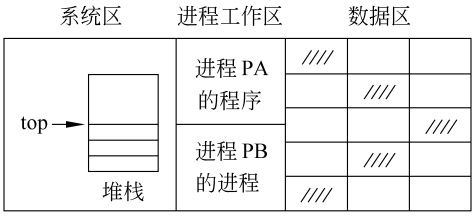

图 3.11　两个进程共享内存

当进程 PA 或 PB 要求使用空数据块时，从堆栈顶部（top 指针所指的位置）取出所需数据块地址。当进程 PA 或 PB 释放数据块时，则把释放的数据块的地址放入堆栈顶部。getspace 为取空数据块地址过程。release(ad)为释放数据块过程，这里，ad 为待释放数据块的地址。如果堆栈非空，进程 PA 或 PB 可以按任意的顺序释放和获取数据块地址。执行 getspace 就是获取一个空数据块地址，而执行 release(ad)就是释放一个地址为 ad 的数据块。然而，由下面的描述可以看到，在进程并发执行时，getspace 或 release(ad)有可能完不成任务。

getspace 和 release(ad)可进一步描述如下：

```
getspace: begin local g
  g←stack[top]
  top←top-1
end
release(ad): begin
  top←top+1
  stack[top]←ad
end
```

设时刻 t0 时 top＝h0，则 getspace 和 release(ad)可能按以下顺序执行。

首先，release(ad)的第一句执行：

```
t0: top←top+1 →top=h0+1;
```

其次，getspace 执行：

```
t1: g←stack[top]→g=stack[h0+1];
t2: top←top-1 →top=h0;
```

最后，release(ad)的第二句执行：

```
t3: stack[top]← ad→stack[h0]←ad;
```

其结果是调用 getspace 的进程取到的是 h0+1 中的一个未定义值，而调用 release(ad) 的进程把释放的空数据块地址 ad 重新放入了 h0 中。

怎样保证上述执行结果的正确性呢？一个明显的答案是：如果把 getspace 和 release(ad) 抽象为两个各以一个动作完成的顺序执行单位，那么执行结果的正确性是可以保证的。

把不允许多个并发进程交叉执行的一段程序称为临界区(critical region)或临界部分(critical section)。

临界区是由属于不同并发进程的程序段共享公用数据或公用数据变量而产生的。例如，上例中的临界区就是因为过程 getspace 和 release(ad)共同访问堆栈中的数据块地址而产生的。临界区不可能用增加硬件的方法解决。

2. 间接制约

一般来说，可以把不允许交叉执行的临界区按不同的公用数据划分为不同的集合。在上例中，按公用数据堆栈划分的临界区集合是{getspace，release}。把这样的集合称为类(class)。显然，对类给定一个唯一的标识名，系统就能够区分它们。在程序的描述中，可用下列标准形式来描述临界区：

```
when <类名> do <临界区> od
```

设类{getspace，release}的类名为 sp，则 getspace 和 release(ad)可重新描述为

```
getspace: when sp do getspce←stack[top]
top←top-1 od
release(ad): when sp do top←top+1
stack[top]←ad od
```

把这种由于共享某一公有资源而引起的在临界区内不允许并发进程交叉执行的现象称为由共享公有资源造成的对并发进程执行速度的间接制约，简称间接制约。这里，“间接”主要是指各并发进程的速度受公有资源制约，而不是进程间直接制约的意思。

这里，受间接制约的类中各程序段在执行顺序上是任意的。

显然，对于每一类，系统要有分配和释放相应的公有资源的管理办法，以制约并发进程。这就是互斥。

3. 什么是互斥

综上所述，可以把互斥定义为：一组并发进程中的多个程序段因共享某一公有资源而导致它们必须以一个不允许交叉执行的单位执行。也就是说，互斥是指不允许两个或两个以上的共享该资源的并发进程同时进入临界区。

这里，考虑类中只有一个元素，也就是只有一个程序段的情况是很有意思的。这时该程序段本身作为公有资源被并发进程共享。一般情况下，作为程序段的一个过程不允许多个进程同时访问它；但是，如果该过程是纯过程，则各并发进程可以同时访问它。什么是纯过程？它是在执行过程中不改变自身代码和执行结果的过程。通常，在计算机系统中，有许多过程(例如编辑程序、编译程序等)是被多个并发进程同时访问的。把一个过程编写成纯过程有利于多个进程共享，但由于编制纯过程必须对有关变量和工作区作相应的处理，从而其

执行效率往往会受到一定的影响。

一组并发进程互斥执行时必须满足如下准则：

(1) 不能假设各并发进程的相对执行速度。各并发进程享有平等、独立地竞争共有资源的权利，而且在不采取任何措施的条件下，在临界区内任一指令结束时，其他并发进程可以进入临界区。

(2) 并发进程中的某个进程不在临界区中时，不阻止其他进程进入临界区。

(3) 并发进程中的若干进程申请进入临界区时，只允许一个进程进入临界区。

(4) 并发进程中的某个进程申请进入临界区后，应在有限时间内得以进入临界区。

这里，准则(1)～(3)保证各并发进程享有平等、独立地竞争和使用公有资源的权利，且保证每一时刻至多只有一个进程在临界区中。准则(4)是并发进程不发生死锁的重要保证。否则，由于某个并发进程长期占有临界区，其他进程就会因为不能进入临界区而处于互相等待状态。

在一组并发执行的进程中，除了因为竞争公有资源而引起的间接制约之外，还存在因为并发进程互相共享对方的私有资源而引起的直接制约。直接制约将使得各并发进程同步执行。下面讨论互斥的实现方法。

3.4.2 互斥锁

在 3.4.1 节中，给出了临界区的描述方法和并发进程互斥执行时必须遵守的准则，但是并没有给出怎样实现并发进程的互斥。人们可能认为，只需把临界区中的各个过程按不同的时间排列，依次调用就行了。但事实上这是不可能的，因为这要求该组并发进程中的每个进程事先知道其他并发进程与系统的动作，由用户程序执行开始的随机性可知，这是不可能的。

一种可能的办法是对临界区加锁以实现互斥。当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否是上锁的。如果该临界区已被锁住，则并发进程要等到该临界区开锁之后才有可能获得临界区。设临界区的类名为 S。为了保证每一次临界区中只能有一个程序段被执行，设锁定位为 key[S]，表示该锁定位属于类名为 S 的临界区。加锁后的临界区程序描述如下：

```
lock(key[S])
<临界区>
unlock(key[S])
```

设 key[S]＝1 时表示类名为 S 的临界区可用，key[S]＝0 时表示类名为 S 的临界区不能用，则解锁程序 unlock(key[S])只用一条语句即可实现：

```
key[S]←1
```

不过，由于 lock(key[S])必须满足 key[S]＝0 时不允许任何进程进入临界区，而 key[S]＝1时仅允许一个进程进入临界区的准则，因而实现起来较为困难。

一种简便的实现方法是

```
lock(x)=begin local v
    repeat
```

```
        v←x
    until v=1
    x←0
end
```

不过，这种实现方法不能保证并发进程互斥执行所要求的准则(3)。因为当同时有几个进程调用 lock(key[S])时，在 x←0 语句执行之前，可能已有两个以上进程由于 key[S]=1 而进入临界区。为了解决这个问题，有些计算机在硬件中设置了“测试与设置”(test and set)指令，从而保证第一步和第二步执行的不可分离性。

这里，有一点需要注意：在系统实现时，锁定位 key[S]总是设置在公有资源对应的数据结构中的。

3.4.3 信号量与 P、V 原语

1. 信号量

尽管用加锁的方法可以实现进程之间的互斥，但这种方法仍然存在一些影响系统可靠性和执行效率的问题。例如，循环测试锁定位将耗费较多的 CPU 计算时间。如果一组并发进程的数量较多，在极端情况下，所有进程可能同时申请进入临界区，每个进程都要对锁定位进行测试，这时系统的开销是很大的。

另外，使用加锁方法实现进程间互斥时，还将导致在某些情况下出现不公平现象。考虑以下进程 PA 和 PB 反复使用临界区的情况。

PA：

```
A: lock(key[S])
    <S>
  unlock(key[S])
  goto A
```

PB：

```
B: lock(key[S])
    <S>
  unlock(key[S])
  goto B
```

假设进程 PA 已通过 lock(key[S])过程进入临界区。显然，在进程 PA 执行 unlock(key[S])过程之前，key[S]=0，而且进程 PB 没有进入临界区的机会。然而，当进程 PA 执行完 unlock(key[S])过程之后，由于紧接着是一条转向语句，进程 PA 将又一次立即执行 lock(key[S])过程。此时，由于 unlock(key[S])过程已将 key[S]的值置为 1，也就是开锁状态，从而进程 PA 又可进入临界区 S。这时，系统将进入死循环状态。只有当进程 PA 执行完 unlock(key[S])过程之后、执行 goto A 语句之前的瞬间发生进程调度，才可能使进程 PA 把处理器转让给进程 PB，进程 PB 才有可能得到执行。然而，这种可能性是非常小的。因此，进程 PB 将处于永久的饥饿(starvation)状态。

怎样解决上述问题呢？首先，必须找到产生上述问题的原因。显然，在用加锁法解决进

程互斥的问题时，一个进程能否进入临界区是依靠进程自己调用 lock 过程测试相应的锁定位。这样，没有获得执行机会的进程当然无法判断，从而出现不公平现象；而获得了执行机会的进程又因为需要测试而损失一定的 CPU 时间。这正如某个学生想使用某个人人都可借用且不规定使用时间的教室时一样，他必须首先申请获得使用该教室的权利，然后再看该教室是不是被锁上了。如果该教室被锁上了，他只好下次再来看该教室的门是否已被打开。这种重复的活动将持续到他进门后为止。从这个例子中可以得到解决加锁法所带来的问题的方法。一种最直观的办法是设置一个教室管理员。如果有学生申请使用教室而未能如愿，教室管理员记下他的名字，等教室门一打开就通知该学生进入。这样，既减少了学生多次来教室检查门是否被打开的时间，又减少了因为学生自发地检查造成的不公平现象（有的学生可能来几十次也进不了教室门；但有的学生可能一次就进去了，或不断地出出进进）。在操作系统中，这个管理员就是信号量（semaphore）。信号量管理相应临界区的公有资源，它代表可用资源实体。

信号量的概念和下面所述的 P、V 原语是荷兰科学家 E.W.Dijkstra 提出的。信号是铁路交通管理中的一种常用方法，铁路交通管理人员利用信号颜色的变化来实现铁路交通管理。在操作系统中，把信号量定义为 sem。它是一个整数。sem 大于或等于 0 时表示可供并发进程使用的资源实体数，sem 小于 0 时则表示正在等待使用临界区的进程数。显然，用于互斥的信号量 sem 的初值应该大于 0。建立一个信号量时必须说明该信号量的意义，并赋予它初始值。然后，还要建立相应的数据结构，以便和等待使用该临界区的进程关联起来。

2. P、V 原语

信号量的数值仅能由 P、V 原语操作来改变（P 和 V 分别是荷兰语 Passeren 和 Verhoog 的第一个字母，相当于英语的 pass 和 increment 的意思）。采用 P、V 原语，可以把类名为 S 的临界区描述为

```
when S do P(sem) 临界区 V(sem) od
```

这里，sem 是与临界区内使用的公用资源有关的信号量。一次 P 原语操作使得信号量 sem 减 1，而一次 V 原语操作使得信号量 sem 加 1。必须强调的是，当某个进程正在临界区内执行时，其他进程如果执行了 P 原语操作，则该进程并不像调用 lock 时那样因进不了临界区而返回到 lock 的起点并等以后重新执行测试，而是在等待队列中等待其他进程执行 V 原语操作释放资源后进入临界区，这时，P 原语的执行才算真正结束。另外，当有好几个进程执行 P 原语操作未通过而进入等待状态之后，如有某个进程执行了 V 原语操作，则等待进程中的一个可以进入临界区，但其他进程必须继续等待。

P 原语操作的主要动作如下：

（1）sem 减 1。

（2）若 sem 减 1 后大于或等于 0，则 P 原语返回，该进程继续执行。

（3）若 sem 减 1 后小于 0，则该进程被阻塞后进入该信号的等待队列，然后转进程调度程序。

P 原语操作的功能框图如图 3.12 所示。

V 原语的操作主要动作如下：

(1) sem 加 1。

(2) 若 sem 加 1 后大于 0，V 原语停止执行，该进程返回调用处继续执行。

(3) 若 sem 加 1 后小于或等于 0，则从该信号的等待队列中唤醒一个等待进程，然后返回原进程继续执行或转进程调度程序。

V 原语操作的功能框图如图 3.13 所示。

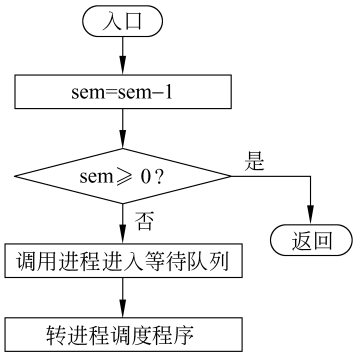

图 3.12 P 原语操作的功能框图

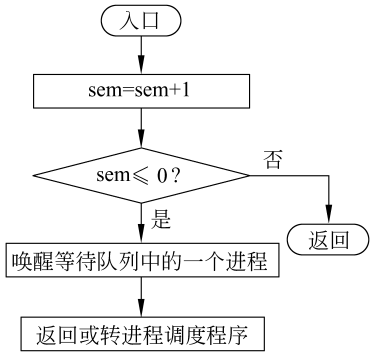

图 3.13 V 原语操作的功能框图

有了加锁法的基础，就容易理解 P、V 操作要以原语实现的原因。否则，如果多个进程同时调用 P 操作或 V 操作，则有可能在 P 操作刚执行 sem 减 1 而未把对应进程送入等待队列时，V 操作就开始执行了。此时，V 操作将无法发现等待进程而返回。因此，P、V 操作都必须以原语实现，且在 P、V 原语执行期间不允许中断发生。

P、V 原语的实现有许多方法。这里介绍一种使用加锁法的软件实现方法，其实现过程描述如下：

P(sem)：

```
begin
    lock(lockbit)
    val[sem]=val[sem]-1
    if val[sem]<0
        保护当前进程 CPU 现场
        当前进程状态置为“等待”
        将当前进程插入信号 sem 的等待队列
        转进程调度程序
    fi
    unlock(lockbit)
end
```

V(sem)：

```
begin
    lock(lockbit)
    val[sem]=val[sem]+1
    if val[sem]⩽0
```

```
        local k
        从 sem 的等待队列中选取一个等待进程,将其指针置入 k 中
        将 k 插入就绪队列
        将进程 k 的状态置为"就绪"
    fi
    unlock(lockbit)
end
```

在以上过程中,lock(lockbit)和 unlock(lockbit)分别为关闭中断和开放中断。

3. 用 P、V 原语实现进程互斥

利用 P、V 原语和信号量 sem,可以方便地实现并发进程的互斥,而且不会产生使用加锁法实现互斥时出现的问题。

设 sem 是用于互斥的信号量,且其初值为 1,表示没有并发进程使用该临界区。由前面的讨论可知,只要把临界区置于 P(sem)和 V(sem)之间,即可实现进程间的互斥。

其实现过程是:当一个进程要进入临界区时,它必须先执行 P 原语操作,将信号量 sem 减 1;当一个进程完成对临界区的操作之后,它必须执行 V 原语操作,释放它占用的临界区。由于信号量初始值为 1,所以,进程在执行 P 原语操作之后将 sem 的值变为 0,表示该进程可以进入临界区。在该进程未执行 V 原语操作之前如果有另一进程要进入临界区,它也应先执行 P 原语操作,从而使 sem 的值变为-1,因此,第二个要进入临界区的进程将被阻塞。直到第一个进程执行 V 原语操作之后,sem 的值变为 0,即可唤醒第二个进程进入就绪队列,经调度后再进入临界区。在第二个进程执行完 V 原语操作之后,如果没有其他进程申请进入临界区,则 sem 又恢复到初始值。

这里有一个要注意的问题,即 sem 信号值为 0 时,如果在临界区等待队列中有一个进程处于等待状态,则该进程会被调度选中执行,但 sem 信号的值不会改变。如果在临界区等待队列中没有等待进程,则 sem 信号的值为 0 时不会有新的进程执行临界区程序。

用信号量实现两并发进程 PA、PB 互斥的描述如下:

设 sem 为互斥信号量,其取值为{1,0,-1}。其中,sem=1 表示进程 PA 和 PB 都未进入类名为 S 的临界区;sem=0 表示进程 PA 或 PB 已进入类名为 S 的临界区,sem=-1 表示进程 PA 和 PB 中,一个进程已进入临界区,而另一个进程等待进入临界区。

PA:

```
P(sem)
<S>
V(sem)
⋮
```

PB:

```
P(sem)
<S>
V(sem)
⋮
```