第3章 进程管理

处理机是系统的重要资源之一，而处理机的管理实际上是进程的管理。在现代计算机系统中，通常以进程的观点来设计和研究操作系统。因此，只有深刻理解进程的概念，才能够很好地理解操作系统各部分的功能和工作原理。

本章首先引入进程的概念，指出其特点，然后逐步介绍进程的管理，包括进程的建立、调度、控制等。

3.1 进程的概念

进程是现代操作系统最重要的概念之一。在多道程序系统中程序并发执行导致其出现了一些与单道程序顺序执行时不同的特征，由此引入了进程的概念。

3.1.1 进程的引入

1. 单道程序的顺序执行

在早期的计算机系统中，只有单道程序执行的功能。也就是说，每一次只允许一道程序运行，这个程序在运行期间将独占整个计算机系统的资源，而且系统按照程序的步骤顺序地执行，在该程序执行完之前，其他程序只能等待。这种程序执行方式称为顺序执行方式。程序的顺序执行具有如下特点：

(1) 顺序性。程序的执行过程是一系列严格按程序规定的状态转移的过程。上一条指令的执行结果是下一条指令的执行开始的充分必要条件。

(2) 封闭性。程序是在封闭的环境下执行的。即程序执行时独占资源，资源的状态(除初始状态外)只有本程序才能改变。程序执行得到的最终结果由初始条件决定，不受外界因素的影响。

(3) 可再现性。程序执行结果与它的执行速度无关。只要输入的条件相同，重复执行时，不论它是从头到尾不停顿地执行，还是“停停走走”地执行，都会得到相同的结果。

程序的顺序执行的特点使系统管理非常方便，程序员检验和校正程序的错误也很容易。然而，系统的资源利用率却非常低，尤其在对外部设备进行操作的时间内，系统处理器都在等待。

2. 多道程序系统中程序的执行

为提高处理机的效率，人们设想让多个程序同时执行。然而，单处理机系统每一时刻只能执行一条指令。如果要同时执行多条指令，必须具有多个处理机或者处理部件，这就是并行结构和并行处理要解决的问题。

能否在单处理机上实现程序的同时执行呢？这就是程序的并发执行问题。并发执行是基于多道程序的一个概念，即让多道程序在计算机中交替地执行，当一道程序不用处理机时，另一道程序就马上使用处理机，从而大大提高了处理机的利用率。虽然在每一时刻仍然只有一条指令在执行，但在计算机的主存储器中同时存放了多道程序，在同一时间间隔内，这些程序在交替地执行。因此，在微观上指令是顺序执行的，而在宏观上程序是并发执行的，从而减少了处理机的等待时间，使得处理机和外设可同时工作，提高了系统的使用效率。

程序的并发执行虽然提高了系统的使用效率，但由于多道程序在主机中并发执行，共享系统资源，因而产生了一些与顺序程序不同的特点：

(1) 间断性。程序在并发执行时，由于它们共享系统资源，并且为完成同一项任务而相互合作，致使在这些并发执行的程序之间形成了相互制约的关系。例如，几个并发程序竞争同一资源(如打印机)，得到资源的程序继续运行，而其他的程序则只有等待，这是间接制约。又如，一个程序请求从磁盘中读入一个文件，它就直接受到系统磁盘管理程序何时完成该请求的制约，只有等到后者读入了指定的文件后，该程序才能继续执行与该文件有关的操作，这是直接制约。由于存在制约，就存在等待。因此，并发程序具有“执行—暂停—执行”这种间断性的活动规律。

(2) 失去封闭性。程序在并发执行时，多个程序共享系统中的各种资源，因而这些资源的状态将由多个程序来改变，致使程序的运行失去封闭性。这样，某个程序在执行时必然会受到其他程序的影响。例如，当处理机这一资源已被某个程序占用时，另一程序必须等待。

(3) 不可再现性。程序在并发执行时，由于失去了封闭性，也将导致其失去可再现性。

例 3-1 设有两道程序 CP 和 PP，它们共享一个变量 n，其初值为 0。CP 程序循环 10 000 次，执行 n=n+1；PP 程序打印 n 的值。CP 和 PP 可分别描述如下：

```
CP()                                PP()
{                                   {
    while(n<10000)                      printf("n= %d\n",n);
        n=n+1;                      }
}
```

显然，如果上例中的 CP 和 PP 程序顺序执行，其执行结果为 n=10000(屏幕显示)。但如果让两个程序段并发执行，程序 CP 和 PP 的执行速度不同，将有可能出现下述几种情况：

(1) 首先程序 CP 抢占了处理机开始执行，然后执行程序 PP，执行结果是 n=10000。

(2) 首先程序 PP 抢占了处理机开始执行，然后执行程序段 CP，执行结果是 n=0。

(3) 如果在某一分时系统中，首先程序 CP 开始执行，执行到某一时间，其时间片用完(假设 CP 执行到 n=2000)，这时程序 PP 也开始执行且抢占了处理机，执行结果为 n=2000 和 n=10000。

(4) 如果将两个程序放在另一个执行速度较快的分时系统中执行，假设 CP 执行到 n=

3000 时,其时间片用完,这时程序 PP 开始执行,其执行结果为 n=3000 和 n=10000。

这说明,程序在并发执行时,有一些程序经过多次执行,虽然它们执行的初始条件相同,但其执行速度不同,结果也不同。即,在多道程序系统中,程序的执行不再具有可再现性,甚至会出现错误的结果。

3. 进程概念的引入

从上述讨论可以看出,在多道程序环境下,程序并发执行,程序的执行具有了许多新的特性,程序与执行结果不再一一对应。在多道程序系统中,一般情况下,并发执行的各程序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(例 3-1 中的程序并发执行出现不同的结果是由于两个程序共享变量 n)。显然,这是程序设计人员不希望看到的。为了在并发执行时不出现错误结果,必须采取某些措施来制约、控制各并发程序的执行速度,这在操作系统程序设计中尤为重要。

为了控制和协调各程序执行过程中对软硬件资源的共享和竞争,在操作系统中引入了进程的概念来反映和刻画系统和用户程序的活动。

3.1.2 进程的定义

1. 进程的定义

进程的概念是 20 世纪 60 年代初期首先在 MIT 的 Mulitics 系统和 IBM 的 TSS/360 系统中引入的。此后,人们对进程下过各式各样的定义。现列举其中几种:

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

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

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

(4) 行为的规则叫程序,程序在处理机上执行的活动称为进程(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(Brinch Hansen)。

以上对进程的定义尽管各有侧重,但本质是相同的,即主要注重进程是一个程序的执行过程这一概念。进程是程序的一次运行活动,即程序是一种静态的概念;而进程是一种动态的概念,它是“活动的”。

进程和程序之间的区别是很微妙的,却非常重要。通过一个类比可以使我们更容易理解这一点。想象一位有一手好厨艺的计算机科学家正在为他的女儿烘制生日蛋糕。他有做生日蛋糕的食谱,厨房里有所需的原料:面粉、鸡蛋、糖、香草汁等。在这个类比中,做蛋糕的食谱就是程序(即用适当形式描述的算法),计算机科学家就是处理机(CPU),而做蛋糕的各种原料就是输入的数据。进程就是厨师阅读食谱、取来各种原料以及烘制蛋糕的一系列动作的总和。

现在假设计算机科学家的女儿哭着跑了进来,说她被一只蜜蜂螫了。计算机科学家就记录下自己照着食谱做到哪儿了(保存进程的当前状态),然后拿出一本急救手册,按照其中的指示处理螫伤。这里,我们看到处理机从一个进程(做蛋糕)切换到另一个高优先级的进程(实施医疗救治),每个进程拥有各自的程序(食谱和急救指示)。当蜜蜂蜇伤处理完之后,

计算机科学家又回来做蛋糕，从他离开时的那一步继续做下去(继续执行做蛋糕程序)。

这里的关键思想是：一个进程是某种类型的一个活动，它有程序、输入、输出及状态。单个处理机被若干进程共享，它使用某种调度算法决定何时停止一个进程的工作，并转而为另一个进程提供服务。

在此给出进程的定义：进程是一个具有独立功能的程序对某个数据集在处理机上的执行过程和分配资源的基本单位。

2. 进程的特征

在操作系统中，进程是进行系统资源分配、调度和管理的最小独立单位，操作系统的各种活动都与进程有关。为了进一步明确进程的概念，下面给出进程的一些突出特征。

1) 动态性

动态性是进程最基本的特征之一。进程是程序的一次运行过程，具有一个从静止到活动的过程，即诞生、运行、消失的过程，有一定的生命周期，它与程序并不一一对应。程序是静态的，只是指令的集合，作为一种文件可长期存放在存储装置中。一个进程可以对应一个程序，或者对应一段程序(一个程序的部分)；一个程序可以对应一个进程，也可以对应多个进程，此时称这个程序被多个进程所共享，可共享的程序代码被称为可重入代码或者纯代码，纯代码在运行过程中不能被改变。

2) 并发性

并发性是指多个进程同时驻留内存，且能在一段时间内交替运行。并发性是进程最基本的特征之一，同时也是操作系统的重要特征。引入进程的目的也正是为了使多个进程能并发运行，而程序是不能并发运行的。

3) 独立性

进程是操作系统中可以独立运行的基本单位，也是分配资源和进行调度的基本单位。进程在获得其必需的资源后即可运行，在不能得到某个资源时便停止运行。在具有并发性的系统中，未建立进程的程序不能作为一个独立单位运行。

4) 异步性

异步性指进程的执行起始时间的随机性和执行速度的独立性。

5) 结构性

为了记录、描述、跟踪和控制进程的变化过程，系统建立了一套重要的数据结构，每个进程有其对应的数据结构。

3.1.3 引入进程的利弊

引入进程是多道程序和分时系统的需要，也是描述程序并发执行活动的需要。在操作系统中，通过为每道程序建立进程，使它们彼此间能够并发执行，从而改善系统资源的利用率，提高系统的吞吐量。因此，目前几乎所有的操作系统中都引入了进程的概念，支持多进程的操作。然而，引入进程也带来如下的问题。

1) 空间开销

系统必须为每个进程建立必需的数据结构和管理数据结构的机构，它们将占据一定的存储器空间。在内存容量较小的情况下，与进程有关的空间开销成为一个包袱，会影响内存

空间使用率。如果内存空间足够大,空间影响就会降低。

2) 时间开销

系统为了管理和协调进程的运行,要不断跟踪进程的运行过程,不断更新有关的系统和进程数据结构,进行进程间的运行工作切换、现场保护等。这些都需要占用处理机的时间,使系统付出时间开销。时间开销与操作系统设计、数据结构的选择以及高速处理机的采用都有直接关系。

3.2 进程控制块和进程的状态

由前面的叙述可知,进程在执行过程中,具有"执行—暂停—执行"这种间断性的活动规律。需要有一个专门的机制能够管理进程的这种变化过程。

3.2.1 进程的状态及其变化

由于各进程在其生命周期内的并发执行及相互制约,使得它们的状态不断发生变化。一般而言,进程具有以下 3 种最基本的状态。

(1) 就绪状态。当进程已分配到除处理机以外的所有必要资源后,只要再获得处理机,便可立即执行,进程这时的状态称为就绪状态。

(2) 运行状态。已经获得处理机及其他的运行资源,正在处理机上运行的进程处于运行状态。

(3) 等待状态。正在执行的进程由于某种运行条件不具备而暂停执行时,在等待某一事件的发生,此时进程处于等待状态,有时也称为阻塞状态。致使进程等待的典型事件有请求 I/O、申请缓冲空间等。

在单处理机系统中,处于运行状态的进程只有一个。正在运行的进程如果因分配给它的时间片到而被暂停运行时,该进程便由运行状态又回到就绪状态;处于就绪状态的进程可以有多个,它们都具备了运行的所有条件,仅仅未获得处理机控制权,如果有多个处理机,这些就绪进程都可以转入运行状态。如果需要等待某一事件的发生而使运行受阻(例如,进程请求访问某种独享资源,而该资源正在被其他进程占用,必须等到该资源被释放),该进程将由运行状态转变为等待状态。当引起阻塞的原因解除后,即回到就绪状态。图 3-1 给出了 3 个基本状态之间的转换关系。

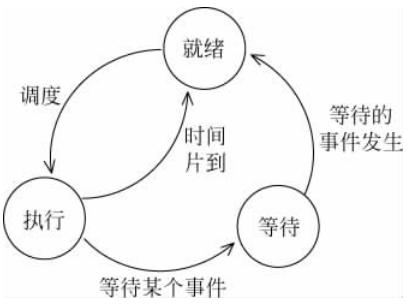

图 3-1 进程状态转换

进程状态转换发生变化的原因和条件归根到底源于进程之间的相互制约关系。对进程状态的转换过程,需要注意如下 3 点:

(1) 进程从等待状态到运行状态,必须经过就绪状态,而不能直接转换到运行状态。这是因为此进程等待的原因解除后,系统中可能有多个进程都处于可运行状态(就绪状态),因此系统必须按照一定的算法选择一个就绪进程占用处理机,这种选择过程被称为进程调度

(schedule)。

(2) 一个进程由运行状态转变为等待状态一般是由运行中的进程自己主动提出的。例如,进程在运行过程中需要某一条件而不能满足时,就自己主动放弃处理机而使进程转入等待状态。

(3) 一个进程由等待状态转变为就绪状态总是由外界事件引起的,而不是由该进程自己引起的。例如某一 I/O 操作完成,由 I/O 结束中断来解除等待此 I/O 完成的进程的等待状态,将其转换为就绪状态。

以上 3 种状态是进程最基本的状态,在实际的操作系统中往往不止这 3 种状态。进程的状态设置和规定与实际的操作系统设计有关。例如,还可以设置自由态、睡眠态、接收态、停止态等。对 3 种基本状态也可以再细分,例如,分为静止就绪、活动就绪、静止等待、活动等待等。这些状态的设立和状态之间的转换均与系统进程的调度需要有关,根据操作系统的设计目标不同而不同。

3.2.2 进程控制块

1. 进程的静态描述

进程既然是一个动态的概念,那么如何表示一个进程,又如何知道进程的存在呢? 显然在系统中需要有描述进程存在和能够反映其变化的物理实体,即进程的静态描述。进程的静态描述由 3 部分组成:

(1) 程序:指进程运行所对应的执行代码。

(2) 数据集合:指程序加工的对象和场所。是进程运行中必需的数据资源,包括对 CPU 占用、存储器、I/O 通道等的需求信息。

(1)和(2)两部分内容与进程的执行有关,大多数操作系统把这两部分内容放在外存中,直到该进程执行时再调入内存。

(3) 进程控制块:是系统为每个进程定义的一个数据结构。它包含了有关进程的描述信息、控制信息和资源信息,是进程动态特征的集中反映。

2. 进程控制块的作用

为了描述和控制进程的运行,系统为每个进程定义了一个数据结构——进程控制块(Process Control Block,PCB),它是操作系统中最重要的记录型数据结构。PCB 中记录了操作系统所需的、用于描述进程的当前情况以及进程控制运行的全部信息。进程控制块的作用是使一个多道程序环境下不能独立运行的程序(含数据)成为一个能独立运行的基本单位,一个能与其他进程并发执行的进程。或者说操作系统是根据 PCB 来对并发执行的进程进行控制和管理的。例如,当操作系统要调度某进程时,从该进程的 PCB 中查出其现行状态及优先级;在调度到某进程后,要根据 PCB 中所保存的处理机状态信息,设置该进程恢复运行的现场,并根据其 PCB 中程序和数据的内存始址,找到其程序和数据;进程在执行过程中,当需要和与之合作的进程实现同步、通信和访问文件时,也需要访问 PCB;当进程因某种原因而暂停执行时,又须将其断点的处理机环境保存在 PCB 中。可见,在进程的整个生命周期中,系统总是通过 PCB 对进程进行控制的,亦即,系统是根据进程的 PCB 而感

知进程的存在的。所以说,PCB是进程存在的唯一标志。

当系统创建一个新进程时,就为它建立了一个PCB;进程结束时又收回其PCB,进程于是也随之消亡。PCB可以被操作系统中的多个模块(如被调度程序、资源分配程序、中断处理程序以及监督和分析程序等)读或修改。因为PCB经常被系统访问,因此,在几乎在所有的多道操作系统中,一个进程的PCB全部或部分常驻内存。

3. 进程控制块中的信息

一般来说,根据操作系统的要求不同,进程的PCB所包含的内容会多少有所不同。但是,下面所示的基本信息是必需的。

1) 描述信息

(1) 进程标识符。在所有的操作系统中,创建一个进程时,系统即为该进程赋予一个唯一的内部标识符,以便系统区别每个进程。

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

(3) 家族关系。在有的系统中,进程之间存在家族关系。PCB中相应的项描述其家族关系。

2) 控制信息

(1) 进程当前状态。说明进程当前处于何种状态。

(2) 进程优先级。是选取进程占用处理机的重要依据。与进程优先级有关的PCB表项有占用CPU时间、进程优先级偏移、占据内存时间等。

(3) 程序开始地址。指出该进程的程序从此内存地址开始执行。

(4) 各种计时信息。给出进程占用和利用资源的有关情况。

(5) 通信信息。记录进程在运行过程中与其他进程通信的有关信息。

3) 资源管理信息

(1) 占用内存大小及其管理用数据结构指针,例如后面介绍的内存管理中所用到的进程页表指针等。

(2) 在某些复杂系统中,还有对换或覆盖用的有关信息,如对换程序段长度、对换外存地址等。这些信息在进程申请、释放内存时使用。

(3) 共享程序段大小及起始地址。

(4) 输入输出设备的设备号,所要传送的数据长度、缓冲地址、缓冲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释放设备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现场保护机构

处理机状态信息主要是由处理机的各种寄存器中的内容组成的。当处理机被中断时,所有这些信息都必须保存在PCB中,以便在该进程重新执行时,能从断点继续执行。这些寄存器包括:

(1) 通用寄存器。又称为用户可视寄存器,它们是用户程序可以访问的,用于暂存信息。在大多数处理机中,有8~32个通用寄存器,在RISC结构的计算机中可超过100个。

(2) 指令计数器。其中存放了要访问的下一条指令地址。

(3) 程序状态字 PSW。其中含有状态信息,如条件码、执行方式、中断屏蔽标志等。

(4) 用户栈指针。每个用户进程都有一个或若干与之相关的系统栈,用于存放过程和系统调用参数及调用地址。

例 3-2 一种用 C 语言描述的 PCB 结构。

```
struct pentry{int  pid;      .          //进程标识符
              int  pprio;               //进程优先级
              char pstate;              //进程状态
              int  pname;               //进程用户标识符
              int  msg;                 //进程通信信息
              int  paddr                //进程对应的程序执行地址
              int  pregs[SIZE];         //现场保护区大小
              …
}pcb[];                                 //定义进程控制块结构数组
```

可见,通过这些表项内容,标识了进程的存在和运行,集中反映了进程的动态特征。由此系统通过 PCB 就可以对进程进行管理和控制。

4. PCB 的组织方式

在一个系统中,通常可拥有数十个、数百个乃至数千个 PCB。为了能对它们加以有效的管理,应该用适当的方式将这些 PCB 组织起来。目前常用的组织方式有两种。

(1) 链接队列方式。处于相同状态的 PCB 组成队列,形成运行、就绪和阻塞队列。每个 PCB 增加一个链指针表项,指向队列中下一个 PCB 的起始地址,系统中设置固定单元指出各单元的头,即每个队列第一个 PCB 的起址。运行队列实际上只有一个成员,用运行队列指针指向它即可。就绪队列的排队原则与调度策略有关。阻塞队列可以有多个,可根据阻塞原因的不同而把处于阻塞状态的进程的 PCB 排成等待 I/O 操作完成的队列和等待分配内存的队列等。图 3-2 给出了一种链接队列的组织方式。

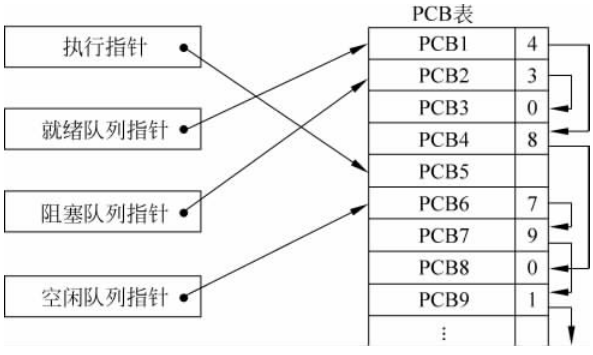

图 3-2 PCB 链接队列示意图

(2) 索引表方式。系统根据所有进程的状态建立几张索引表,例如就绪索引表、阻塞索引表等,并把各索引表在内存的首地址记录在内存的一些专用单元中。在每个索引表的表目中,记录具有相应状态的某个 PCB 在 PCB 表中的地址。图 3-3 给出了索引方式的 PCB 组织。

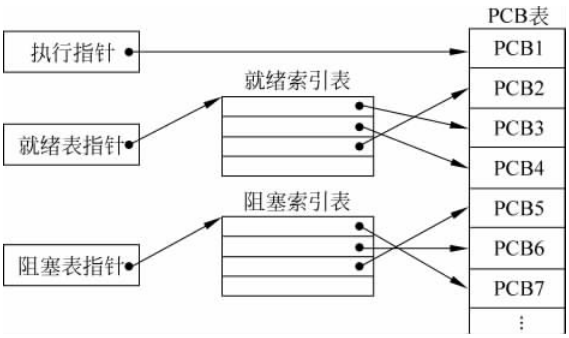

图 3-3 按索引表方式组织 PCB

3.3 进程的控制

进程在从创建到消失的整个生命周期中状态是不断发生变化的。那么,如何控制进程状态的变化呢?如何创建和撤销一个进程呢?

操作系统的进程控制机构控制进程的状态转换。进程的控制机构首先表现在建立、撤销、阻塞、唤醒等方面。通常操作系统内核提供了称作原语的具有特定功能的程序段来完成进程的建立、撤销进程以及完成进程各状态间的转换。

原语被认为是机器语言的延伸,是在系统核心态下执行的,由一条或若干条机器指令组成的具有特定功能的程序段。它一旦被启动,在执行期间是不可中断的。操作系统原语对用户是透明的,一般不允许用户直接使用,以避免对操作系统内核的干扰和破坏。但随着系统的发展,为方便系统程序员使用,有的原语被作为一种特殊的系统调用,既提供给系统进程,也提供给用户进程,通过系统调用方式使用。

3.3.1 进程的创建原语

1. 进程图

一个进程可以创建若干个新进程,新创建的进程又可以继续创建进程,这个创建过程形成了一种树形结构,在操作系统中称为进程图(process graph)。进程图是一棵有向树,其节点代表进程,分枝代表创建。若进程 A 创建了进程 B,称 A 是 B 的父进程(parent process),而 B 称为 A 的子进程(progeny process)。进程树形成了一个进程"家族",根节点为该家族的"祖先"(ancestor)。必须注意,在进程图中,进程 A 创建了进程 B,但并不意味着只有进程 A 执行完以后进程 B 才能执行,而是 A 和 B 可以并发执行。

了解进程间的这种关系是十分重要的。因为子进程可以继承父进程所拥有的资源,例如,继承父进程打开的文件,继承父进程所分配到的缓冲区,等等。当子进程被撤销时,应将其从父进程那里获得的资源归还给父进程。此外,在撤销父进程时,也必须同时撤销其所有的子进程。为了标识进程之间的家族关系,在 PCB 中设置了家族关系表项,以标明自己的父进程及所有的子进程。

2. 引起创建进程的事件

在多道程序环境中，程序只有成为进程时才能在系统中运行。因此，为使程序能运行，就必须为它创建进程。导致一个进程创建另一个进程的典型事件可有以下 4 类：

(1) 用户登录。在分时系统中，用户在终端输入登录命令后，如果是合法用户，系统将为该终端建立一个终端进程，并把它插入就绪队列中。

(2) 作业调度。在批处理系统中，当作业调度程序按一定的算法调度到某作业时，便将该作业装入内存，为它分配必要的资源，并立即为它创建主进程，再插入就绪队列中。

(3) 提供服务。当运行中的用户程序提出某种请求后，系统将专门创建一个进程来提供用户所需要的服务，例如，用户程序要求进行文件打印，操作系统将为它创建一个打印进程，这样，不仅可使打印进程与该用户进程并发执行，而且还便于计算出为完成打印任务所花费的时间。

(4) 应用请求。在上述 3 种情况下，都是由系统内核创建一个新进程；而第 4 类事件则是基于应用进程的需求，由它自己创建一个新进程，以便使新进程以并发运行方式完成特定任务。例如，某应用程序需要不断地从键盘终端读入用户输入的数据，继而又要对输入数据进行相应的处理，然后再将处理结果以表格形式在屏幕上显示。该应用进程为使这几个操作能并发执行，以加速任务的完成，可以分别建立键盘输入进程、数据处理进程和表格输出进程。

3. 进程创建原语

一旦操作系统发现了要求创建新进程的事件后，便调用进程创建原语按下述步骤创建一个新进程。

(1) 申请空白 PCB。为新进程申请获得唯一的数字标识符，并从 PCB 集合中索取一个空白 PCB。

(2) 为新进程分配资源。为新进程的程序和数据以及用户栈分配必要的内存空间。显然，此时操作系统必须知道新进程所需内存的大小。对于批处理作业，其大小可在用户提出创建进程要求时提供。若是为应用进程创建子进程，也应是在该进程提出创建进程的请求中给出所需内存的大小。对于交互型作业，用户可以不给出内存要求而由系统分配一定的空间。如果新进程要共享某个已在内存的地址空间(即已装入内存的共享段)，则必须建立相应的链接。

(3) 初始化进程控制块。PCB 的初始化包括：①初始化标识信息。将系统分配的标识符和父进程标识符填入新 PCB 中。②初始化处理机状态信息。使程序计数器指向程序的入口地址，使栈指针指向栈顶。③初始化处理机控制信息。将进程的状态设置为就绪状态或静止就绪状态。对于优先级，通常是将它设置为最低优先级，除非用户以显式方式提出高优先级要求。

(4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

3.3.2 进程的撤销原语

1. 引起进程撤销的事件

1) 正常结束

在任何计算机系统中,都应有一个用于表示进程已经运行完成的指示。例如,在批处理系统中,通常在程序的最后安排一条 Holt 指令来终止程序的执行。当程序运行到 Holt 指令时,将产生一个中断,通知操作系统本进程已经完成。在分时系统中,用户可利用 Logs off 表示进程运行完毕,此时同样可产生一个中断,通知操作系统进程已运行完毕。

2) 异常结束

在进程运行期间,由于出现某些错误和故障而迫使进程终止。这类异常事件很多,常见的有:①越界错误。这是指程序所访问的存储区已越出该进程的区域。②保护错。进程试图访问一个不允许访问的资源或文件,或者以不适当的方式进行访问,例如,进程试图去写一个只读文件。③非法指令。程序试图执行一条不存在的指令。出现该错误的原因可能是程序错误地转移到数据区,把数据当成了指令。④特权指令错。用户进程试图执行一条只允许操作系统执行的指令。⑤运行超时。进程的执行时间超过了指定的最大值。⑥等待超时。进程等待某事件的时间超过了规定的最大值。⑦算术运算错。进程试图执行一个被禁止的运算,例如被 0 除。⑧I/O 故障。这是指在 I/O 过程中发生了错误等。

3) 外界干预

外界干预并非指在进程运行中出现了异常事件,而是指进程应外界的请求而终止运行。这些干预有:①操作员或操作系统干预。由于某种原因,例如发生了死锁,由操作员或操作系统终止该进程。②父进程请求。由于父进程具有终止自己的任何子孙进程的权利,因而当父进程提出终止某个子孙进程的请求时,系统将终止该进程。③父进程终止。当父进程终止时,操作系统也将其所有子孙进程终止。

2. 进程撤销原语

如果系统中发生了上述要求撤销进程的某事件后,操作系统便调用进程撤销原语,按下述过程撤销指定的进程。

(1) 根据被撤销进程的标识符,从 PCB 集合中检索出该进程的 PCB,从中读出该进程的状态。

(2) 若被撤销进程正处于执行状态,应立即终止该进程的执行,并置调度标识为真,用于指示该进程被撤销后应重新进行调度。

(3) 若该进程还有子孙进程,还应将其所有子孙进程予以撤销,以防它们成为不可控的进程。

(4) 将被撤销进程所拥有的全部资源归还给其父进程或者归还给系统。

(5) 将被撤销进程(它的 PCB)从所在队列(或链表)中移出,等待其他程序使用。

3.3.3 进程的阻塞与唤醒原语

1. 引起进程阻塞和唤醒的事件

下述几类事件会引起进程阻塞或被唤醒。

1) 请求系统服务

当正在执行的进程请求操作系统提供服务时,由于某种原因,操作系统并不能立即满足该进程的要求时,该进程只能转变为阻塞状态来等待。例如,一个进程请求使用某资源,如打印机,由于系统已将打印机分配给其他进程而不能分配给请求进程,这时请求进程只能被阻塞,仅在其他进程释放打印机的同时,才将请求进程唤醒。

2) 启动某种操作

当进程启动某种操作后,如果该进程必须在该操作完成之后才能继续执行,则必须先使该进程阻塞,以等待该操作完成。例如,一个进程启动了某 I/O 设备,如果只有在 I/O 设备完成了指定的 I/O 操作任务后,进程才能继续执行,则该进程在启动了 I/O 操作后,便自动进入阻塞状态去等待。在 I/O 操作完成后,再由中断处理程序或中断进程将该进程唤醒。

3) 新数据尚未到达

对于相互合作的进程,如果其中一个进程需要先获得另一(合作)进程提供的数据才能运行以对数据进行处理,则只要其所需数据尚未到达,该进程就只能等待。例如,有两个进程,进程 A 用于输入数据,进程 B 对输入数据进行加工。假如 A 尚未将数据输入完毕,则进程 B 将因没有所需的数据而等待;一旦进程 A 把数据输入完毕,便可唤醒进程 B。

4) 无新工作可做

系统往往设置一些具有某个特定功能的系统进程,每当这种进程完成任务后,便把自己阻塞起来以等待新任务到来。例如,系统中的发送进程,其主要工作是发送数据,若已有的数据已全部发送完成而又无新的发送请求,这时发送进程将使自己进入等待状态,仅当又有进程提出新的发送请求时,才将发送进程唤醒。

2. 进程阻塞原语

当发生上述某事件时,正在执行的进程由于无法继续执行,于是便通过调用阻塞原语把自己阻塞。可见,进程的阻塞是进程自身的一种主动行为。进入阻塞时,由于此时该进程还处于执行状态,所以应先中断处理机和保存该进程的 CPU 现场。然后把该进程控制块中的现行状态由执行改为阻塞,并将其 PCB 插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列,则应将本进程插入具有相同事件的阻塞(等待)队列。最后,转调度程序进行重新调度,将处理机分配给另一就绪进程。这里,转进程调度是很重要的,否则,处理机将会出现空转而浪费资源。

3. 进程唤醒原语

当被阻塞进程所期待的事件出现时,如 I/O 完成或其所期待的数据已经到达,则由有关进程(例如,用完并释放了该 I/O 设备的进程)调用唤醒原语 wakeup(),将等待该事件的进程唤醒。唤醒原语执行的过程是:首先把被阻塞的进程从等待该事件的阻塞队列中移

出,将其 PCB 中的现行状态由阻塞改为就绪,然后再将该 PCB 插入到就绪队列中。在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。

应当指出,阻塞原语和唤醒原语是一对作用刚好相反的原语。因此,如果在某进程中调用了阻塞原语,则必须在与之相合作的另一进程中或其他相关的进程中安排唤醒原语,以唤醒被阻塞进程;否则,被阻塞进程将会因不能被唤醒而长久地处于阻塞状态,从而再无机会继续运行。

3.4 进程同步

在操作系统中引入进程后,虽然提高了资源利用率和系统吞吐量,但是在进程并发执行时,由于资源共享和进程的合作,使同一系统中的进程之间可能产生两种形式的制约关系,即直接制约和间接制约,而这两种关系通常表现在两类问题上:同步和互斥。进程同步机构的主要任务是使并发执行的诸进程之间能有效地共享资源和相互合作,从而使程序的执行具有可再现性。

3.4.1 互斥

并发进程可以共享系统中的各种资源,但是系统中某些资源具有一次仅允许一个进程使用的属性,这样的资源称为临界资源(critical resource)。例如,一台打印机,若让多个进程任意使用,那么很容易发生多个进程的输出结果交织在一起的混乱情况,解决这一问题唯一的办法就是一个进程提出打印申请并得到许可后,打印机一直被它单独占用。如果在此过程中,另一进程也提出申请,那么它必须等待前一进程释放了打印机以后才可使用。

系统中有很多的物理设备属于临界资源,如卡片输入输出机、打印机、磁带机等,不仅硬件可以是临界资源,软件中的变量、数据、表格都可以是临界资源。下面通过一个例子来说明临界资源的概念。

例 3-3 假设在一个飞机售票系统中,某一时刻数据库中关于某一航班的机票数量 counter=5。某一窗口的售票进程执行的一条操作语句是 counter=counter-1,而另一窗口退票进程执行的一条操作语句是 counter=counter+1。

用高级语言书写的语句 counter=counter+1 和 counter=counter-1 所对应的汇编语言指令如下:

```
LOAD  A, counter;          LOAD  B, counter;
ADD   A, 1;                SUB   B, 1;
STORE A,  counter;         STORE B,  counter;
```

如果让售票进程和退票进程顺序执行,其结果是正确的;但如果并发执行,就会出现差错。问题就在于这两个进程共享了变量 counter。

如果退票进程先执行左列的 3 条机器语言语句,然后售票进程再执行右列的 3 条语句,则最后共享变量 counter 的值仍为 5;反之,如果让售票进程先执行右列的 3 条语句,然后再让退票进程执行左列的 3 条语句,counter 值也还是 5。但是,如果按下述顺序执行:

```
LOAD  A, counter;   (A = 5)
```

```
ADD   A, 1;          (A = 6)
LOAD  B, counter;    (B = 5)
SUB   B, 1;          (B = 4)
STORE A, counter;    (counter = 6)
STORE B, counter;    (counter = 4)
```

则 counter 值是 4，显然不是用户想要的值。读者可以自己试试，倘若再将两段程序中各语句交叉执行的顺序改变，又可能得到 counter＝6 的答案，这表明程序的执行已经失去了可再现性。为了预防产生这种错误，解决此问题的关键是把变量 counter 作为临界资源处理，即，令售票进程和退票进程共享同一变量 counter 的那段代码不能交叉执行。

进程中访问临界资源的那段代码称为关于该临界资源的临界区(critical section)。如上例中的 counter＝counter＋1 和 counter＝counter－1 语句。涉及同一临界资源的不同进程中的临界区称为同类临界区。以后不加特别说明，均指同类临界区。

有了临界区的概念后，进程的互斥就可以描述为：一组并发进程中的两个或多个程序段，因共享某一公有资源而使得这组并发进程不能同时进入临界区的关系称为进程的互斥。

由前述可知，不论硬件临界资源还是软件临界资源，系统中多个进程必须互斥地对它们进行访问。显然，若能保证进程互斥地进入自己的临界区，就能实现诸进程对临界资源的互斥访问。为此，必须有软件方法或同步机构来协调它们。该算法或同步机构应遵循下述调度准则：

(1) 独立平等。不能假设各并发进程的相对执行速度。即各并发进程享有平等的、独立的竞争共享资源的权利。

(2) 空闲让进。并发进程中的某个进程不在临界区时，它不阻止其他进程进入临界区。

(3) 互斥进入。并发进程中的若干个进程申请进入临界区，只能允许一个进程进入，以保证临界资源的互斥使用。

(4) 让权等待。当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”。

(5) 有限等待。并发进程中的某个进程从申请进入临界区时开始，应在有限的时间内进入临界区，以免进程陷入“死等”。

这里，准则(4)遵循了“尽可能提高 CPU 的有效利用率”的操作系统设计目标；准则(5)是并发进程不发生死锁(关于死锁，将在 4.5 节中介绍)的重要保证。

3.4.2 进程的同步

在并发系统中，进程之间除了对公有资源的竞争而引起的间接制约之外，还存在着直接的制约关系，现在结合下例来讨论这类制约问题。

例 3-4 在控制测量系统中，数据采集任务反复把所采集的数据送入一个单缓冲区；计算任务不断从该单缓冲区中取出数据进行计算。它们之间具有相互的制约关系。即，数据采集进程未把数据放入缓冲区，缓冲区空时，计算进程不应执行取数的过程；同样，当缓冲区满时，计算进程还没有取走一个数据时，数据采集进程不能执行放数的过程。如果不采取任何制约机制，则数据采集过程(collection)和计算过程(calculate)相应的程序段分别描述如下：

```
var buf;                          //定义一个全局缓冲区
int flag = 0;                     //定义一个缓冲区状态标志,0 表示空,1 表示满
collection()                      //数据采集过程向缓冲区送入数据
{
    while (TRUE)
    {
        采集数据;
        while(flag == 1);         //重复测试缓冲区是否满
            将采集的数据放入 buf;
        flag = 1;
    }
}
calculate()                       //计算过程从缓冲区中取走数据
{
    while(TRUE)
    {
        while(flag == 0);         //重复测试缓冲区是否空
        从 buf 中取出数据;
        flag = 0;
        计算处理;
    }
}
```

为了简化问题,在此假设不考虑共享变量 flag 的互斥访问。

显然,上述进程的并发执行会造成 CPU 执行时间的极大浪费(因为其中包含两处反复测试的语句),这是操作系统设计不允许的。由于数据采集任务和计算任务在执行过程中存在相互的制约关系,造成了这种浪费。这种现象在多道操作系统和用户进程中大量存在。

我们把异步环境下的一组并发进程,在某些程序段上需互相合作、互相等待,使得各进程在某些程序段上必须按一定的顺序执行的制约关系称为进程间同步。具有同步关系的一组并发进程称为合作进程。

无论是互斥还是同步,都是在执行的时间顺序上对并发进程的操作加以某种限制。对于互斥的进程,它们各自单独执行时都是正确的,但在临界区内不能混在一起交替执行,需互斥地执行,至于哪个进程先进入临界区则无所谓。而对于同步的进程,各自单独执行会产生错误,必须互相配合,共同推进,各合作进程对公共变量的那部分操作必须严格地按照一定的先后顺序执行。由此可见,互斥和同步对操作时间顺序所加的限制是不同的。

3.4.3 同步机构

从以上讨论可知,为了保证进程间的正确执行,操作系统中必须引入一种机制来控制进程间的互斥和同步关系,以保证进程执行结果的可再现性。系统中用来实现进程间同步与互斥的机构统称为同步机构。大多数同步机构采用一个物理实体,如锁、信息量等,并提供相应的原语。系统通过这些同步原语来控制对共享资源或公共变量的访问,以实现进程间的同步与互斥。

1. 加锁/开锁原语

3.4.2 节中,给出了临界区的描述方法和并发进程互斥执行时所必须遵守的准则,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，再依次调用就行了。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，则该进程要等到该临界区开锁之后才有可能获得临界区。为此，操作系统通常提供加锁/开锁原语来保证进程的互斥执行。

用一个变量 ω 来代表某种临界资源的状态。$\omega=1$ 表示某资源可用，可进入临界区；$\omega=0$ 表示资源正在被使用(临界区正在被执行)。

加锁原语 LOCK(ω)定义如下：

(1) 测试 ω 是否为 1。

(2) 若 $\omega=1$，则 $0\rightarrow\omega$。

(3) 若 $\omega=0$，则返回(1)。

开锁原语 UNLOCK(ω)只有一个动作，即 $1\rightarrow\omega$。

利用加锁/开锁原语，可以很方便地实现进程互斥。当某进程要进入临界区时，首先执行 LOCK(ω)原语。这时，若 $\omega=1$，表示没有别的进程进入此临界资源的临界区，于是它可进入并同时设置 $\omega=0$，禁止其他进程的进入；若 $\omega=0$，则表示有进程正在访问此临界资源，它需循环测试等待。当一个进程退出临界区时，必须执行 UNLOCK(ω)原语，否则任何进程，包括它自己，都无法再使用该共享资源。加锁后的临界区程序描述如下：

```
Pro()
{
    ⋮
    LOCK(ω)
    <临界区>
    UNLOCK(ω)
    ⋮
}
```

加锁/开锁原语可以用关中断的方式实现，在进入锁测试之前关闭中断，直到完成锁测试并加锁之后才开中断。加锁/开锁还可在不同计算机上用不同的硬件指令实现。加锁/开锁机制的优点是简单、易实现。其缺点是循环测试锁定位将损耗较多的 CPU 时间，不能遵循“让权等待”的准则，而使进程陷入“忙等”。

2. 信号量和 P、V 原语

1) 信号量

信号是铁路交通管理中的一种常用设备，交通管理人员利用信号颜色的变化来实现交通管理。在操作系统中，利用信号量(semaphores)来表征一种资源或状态，通过对信号量值的改变来表征进程对资源的使用状况，或判断信号量的值控制进程的状态。

1965 年荷兰科学家 E. W. Dijkstra 提出了支持进程互斥和同步管理的信号量技术方案。实际上他定义的信号量是一个整型变量，具有两个基础的原语，并用荷兰语命名为

Prolangen(降低)和 Verhogen(升起)操作,简称 P、V 操作。

信号量 S 定义如下:

(1) S 是一个整型变量而且初值非负。

(2) 对信号量仅能实施 P(S)操作和 V(S)操作,也只有这两种操作才能改变 S 的值。

(3) 每一个信号量都对应一个(空或空非的)等待队列,队列中的进程处于等待状态。

2) P(S)、V(S)原语

P 原语操作的主要动作如下:

(1) S 减 1。

(2) 若 S 减 1 后仍大于或等于零,则进程继续执行。

(3) 若 S 减 1 后小于零,则该进程被阻塞并进入该信号的等待队列中,然后转进程调度。

V 原语操作的主要动作如下:

(1) S 加 1。

(2) 若相加结果大于零,进程继续执行。

(3) 若相加结果小于或等于零,则从该信号的等待队列中唤醒一个等待进程,然后再返回原进程继续执行或转进程调度。

需要指出的是,P、V 操作具有严格的不可分割性,这包含两层含义:

(1) 由于信号量是系统中的公共变量,它可由若干进程所访问,因此,P、V 操作的执行绝对不允许被中断,以保证在任一时刻只能有一个进程对某一信号量进行操作。换言之,对某一信号量的操作必须是互斥的。

(2) P、V 操作是一对操作,若有对信号量 S 的 P 操作,必须也有对信号量 S 的 V 操作,反之亦然。

关于 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(S)
{
    lock(1ockbit);              //封锁中断
    S = S - 1;
    if S < 0
        block(S,L);             //将当前进程阻塞,插入 S 的等待队列
    unlock(1ockbit);            //开放中断
}

V(S):
{
    lock(lockbit);              //封锁中断
    S = S + 1;
    if S <= 0
        wakeup(S,L);            //将 S 的等待队列中的某一进程唤醒
    unlock(lockbit);            //开放中断
}
```

3）P、V 操作的物理意义

在共享同一类资源的具有相互合作关系的进程之间，信号量的初值用来表示系统中同类资源的可用数目。因此，当 $S=0$ 时，表示没有空闲的该类资源可用；$S<0$ 时，其绝对值表示因请求该类资源而被阻塞的进程数；每执行一次 P 操作意味着请求分配一个单位的某类资源，因此描述为 $S=S-1$；若 $S<0$ 表示已无该类资源可供分配，因此把该进程排列到与该 S 相关的等待队列中。进程使用完某类资源必须执行一次 V 操作，意味着释放一个单位的该类资源，因此描述为 $S=S+1$；若 $S\leqslant 0$ 表示已有进程在等待该类资源，因此唤醒等待队列中的第一个或优先级最高的进程，允许其使用该类资源。

在具有相互合作的同步关系的进程之间，信号量还可代表合作进程之间的消息，每一个 P 操作意味着等待合作进程发来一个消息（或信号），每一个 V 操作表示向合作进程发送一个消息。

现代操作系统中，针对不同的信号对象所采用的 P、V 操作含义和概念有了改变，例如，P 操作称为 down 和 wait 操作，V 操作称为 up 和 signal 操作。

3. 管程

信号量机制是解决进程互斥、同步问题的有效工具，但前提是信号量设置、其初值的确定以及相关进程中安排 P、V 操作的位置必须正确，否则同样也会造成与时间有关的错误，有时甚至造成死锁。Dijkstra 于 1971 年提出，把所有进程对某一临界资源的互斥、同步操作都集中起来，构成所谓的“秘书”进程。1975 年，Hansen 和 Hoare 又把“秘书”进程思想发展为管程概念，把并发进程间的互斥、同步操作分别集中于相应的管程中。

1）管程的组成

如图 3-4 所示，管程是由局部于自己的若干公共变量及其说明和所有访问这些公共变量的过程所组成的软件模块或软件包。

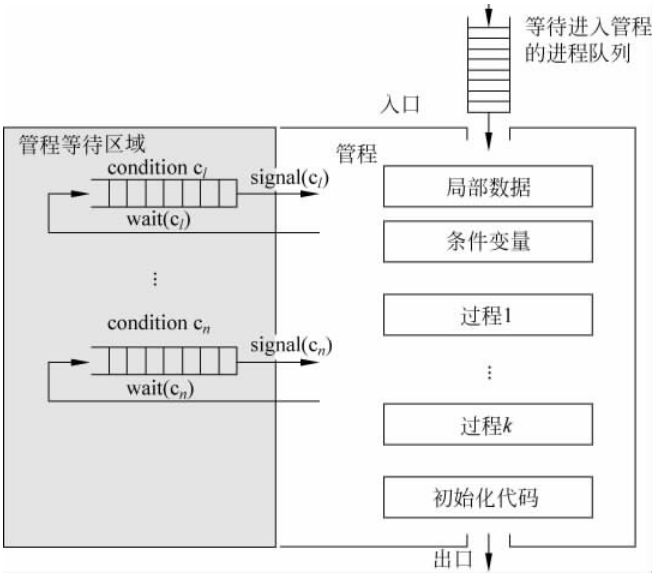

图 3-4 管程结构

管程有3个组成部分：

(1) 局部于管程的共享变量说明(数据结构定义)。这些共享数据表示相应资源的状态；局部于管程的数据结构仅能被局部于管程的过程所访问；局部于管程的过程只能访问管程内部的数据结构。管程相当于围墙，所有进程要访问临界资源时，必须进入管程。

(2) 对数据结构进行操作的一组过程。是每个过程完成前关于上述数据结构的某种规定的操作。

(3) 对局部于管程的数据设置初始值等语句。

2) 管程的基本形式

管程的基本形式如下：

```
TYPE <管程名> = MONITOR
  variable;
  condition;
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
    end;
  …
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
    end;
  begin
    <管程的局部数据初始化语句>;
  end;
```

3) 实现管程的3个关键问题

实现管程时必须考虑3个关键问题，即互斥、同步和条件变量。

(1) 互斥。管程的执行是互斥的，以保证进程互斥地访问临界资源。当几个进程都需调用某一管程时，仅允许一个进程调用进入管程，而其他调用者必须等待。管程的互斥由编译器负责，编译器知道管程的特殊性(只有部分语言支持)。调用管程的程序员无须知道编译器是如何实现互斥的，只需知道将所有的临界区转换成管程的过程即可。

(2) 同步。在管程中必须设置两个同步操作原语 wait 和 signal。当进程通过管程请求访问共享数据而未能满足时，管程便调用 wait 原语使该进程阻塞，并释放管程，此时其他进程可使用该管程。当另一进程访问完该共享数据且释放后，管程便调用 signal 原语，唤醒等待队列中的队首进程。

(3) 条件变量。为了区别等待的不同原因，管程又引入了条件变量。不同的条件变量对应不同原因的进程阻塞等待队列，初始时为空。在条件变量上能作 wait 和 signal 原语操作，若条件变量名为 c，则调用同步原语的形式为 wait. c 和 signal. c。(此处 wait 只是使进程等待，并不改变 c 的值，同理，signal 只是唤醒等待的进程，注意它们与 P、V 的区别)。

3.4.4 同步机构应用

由前述可知，信号量 S 是一个整数。在 S 大于或等于 0 时代表可供并发进程使用的资

源实体数，但 S 小于 0 时则表示正在等待使用该类资源的进程数。因而建立一个信号量必须说明信号量所代表的意义，并赋初值以及建立相应的数据结构，以便指向那些等待使用该临界区的进程。显然，用于互斥的信号量 S 的初值应为 1，而用于同步的信号量的初值应大于或等于 0。

1. 用信号量实现进程互斥

利用 P、V 原语和信号量，可以方便地解决并发进程的互斥问题。对于一组具有互斥关系的进程，只须设置一个互斥信号量 mutex，在临界区的前后加入 P、V 原语即可。

例 3-5 用信号量实现两个并发进程 P_A、P_B 互斥的描述。

由于信号量初始值为 1，表示没有何进程进入临界区，当某一进程进入临界区之前，首先执行 P 原语操作之后将 mutex 的值变为 0，表示已有进程可以进入临界区。这时如果有进程要进入临界区，首先也必须执行 P 原语操作将 mutex 的值变为－1，该进程将阻塞。以此类推，在第一个进程退出临界区之前，其他任何进程都不能进入临界区。直到第一个进程执行完临界区操作，然后执行 V 原语操作之后，才可唤醒某个等待进程进入就绪队列，经调度后再进入临界区。

```
semp mutex = 1;      //互斥信号量
PA()                                  PB()
{                                     {
    ⋮                                     ⋮
    P(mutex);                             P(mutex);
    临界区操作;                           临界区操作;
    V(mutex);                             V(mutex);
    ⋮                                     ⋮
  }                                     }
```

注意：利用 P、V 原语和信号量机制实现进程间的互斥执行，则 P、V 操作必须成对出现在同一个进程里，如果丢失 V 操作将会导致一些进程永远不会被唤醒，如果丢失 P 操作将不能保证临界资源的互斥使用。

2. 用信号量实现进程同步

用信号量实现一组合作进程间的同步执行，通常首先设立与进程执行条件有关的信号量，然后为信号量赋初值，最后利用 P、V 原语规定各进程的执行顺序。

例 3-6 用 P、V 原语实现例 3-4 中的数据采集进程和计算进程的同步执行。

对于数据采集进程，每次放数之前必须申请一个空的缓冲区，因此，为进程 collection 设置一个信号量 Bufempty，代表缓冲区是否为空（可用），其初始值为 1。而对于计算进程 calculate，每次取数之前必须申请一个装满数据的缓冲区，因此，为进程 calculate 设置一个信号量 Buffull，代表缓冲区是否装满数据，其初始值为 0。其相应的程序段分别描述如下：

```
var buf;                          //定义一个全局缓冲区
semp Bufempty = 1;                //设置信号量 Bufempty,表示缓冲区是否为空
semp Buffull = 0;                 //设置信号量 Buffull 表示缓冲区是否装满数据
collection()                      //数据采集进程向缓冲区送数
{
```

```
    while (TRUE)
    {
        采集数据;
        P(Bufempty);                //申请一个空的缓冲区
        将采集的数据放入 buf;
        V(Buffull);                 //释放一个满的缓冲区
    }
}
calculate()                         //计算进程从缓冲区中取走数据
{
    while(TRUE)
    {
        P(Buffull);                 //申请一个满的缓冲区
        从 buf 中取出数据;
        V(Bufempty);                //释放一个空的缓冲区
        计算处理;
    }
}
```

数据采集进程 collection 在送数之前,首先执行 P(Bufempty)操作,申请一个空的缓冲区。执行 P 操作后,若 Bufempty<0,表示没有足够的空缓冲区,进程 collection 阻塞,否则表示可以把数据送入 buf。把数据送入 buf 后,执行 V(Buffull),释放一个满的缓冲区,表示 buf 中有数可取。计算进程 calculate 在取数前首先执行 P(Buffull),申请一个满的缓冲区,若 Buffull<0,表示缓冲区空,进程 calculate 阻塞自己,否则取走数据。计算进程 calculate 取走数据后,执行 V(Bufempty),表示 buf 已空,唤醒进程 collection 送数。

3. 利用信号量实现前趋关系

前趋图(precedence graph)是一个有向无环图,记为 DAG(Directed Acyclic Graph),用于描述进程之间执行的前后关系。图中的每个节点可用于描述一个程序段或一个进程乃至一条语句;节点间的有向边则用于表示两个节点之间存在的偏序(Partial Order)关系或前序关系(Precedence Relation),记作 $P_i \rightarrow P_j$,表示 P_i 执行完以后,P_j 才可以开始执行,称 P_i 是 P_j 的直接前趋,P_j 是 P_i 的直接后继。

例 3-7 对于下述 4 条语句的程序段:

```
P1: a = x + 2
P2: b = y + 4
P3: c = a + b
P4: d = c + b
```

如果建立对应的 4 个进程 P_1、P_2、P_3、P_4,可以看出,P_3 必须在 a 和 b 被赋值后方能执行;P_4 必须在 P_3 之后执行;但 P_1 和 P_2 的执行先后则没有限制,因为它们之间互不依赖。因此,在本例中存在下述的前趋关系:$P_1 \rightarrow P_3$,$P_2 \rightarrow P_3$,$P_3 \rightarrow P_4$,可画出如图 3-5 所示的前趋图。

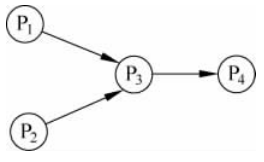

图 3-5 4 条语句的前趋图

信号量机制也可以用来控制程序或语句之间的前趋关系。根据前趋图,对每一对具有前趋关系的进程,设置一个公

用的信号量，并赋初值为0，就可在程序中适当的地方通过该信号量的P、V操作描述并控制这种前趋关系。

对图3-5所示的一组合作进程P_1、P_2、P_3、P_4，为确保这4个进程的执行顺序，设置3个同步信号量a_1、a_2、a_3，分别表示P_1、P_2、P_3是否执行完成，其初值均为0（因为进程P_4没直接后继，也就是说没有进程在等待它完成，所以不需要设置a_4）。这4个进程的同步描述如下：

```
semp a1 = 0;                    //信号量,表示进程 P1 是否执行完成
semp a2 = 0;                    //信号量,表示进程 P2 是否执行完成
semp a3 = 0;                    //信号量,表示进程 P3 是否执行完成
P1()
{
    a = x + 2;                  //执行进程的主体代码
    V(a1);                      //向直接后继 P3 发送信号,表示进程 P1 执行完毕
}
P2()
{
    b = y + 4;                  //执行进程的主体代码
    V(a2);                      //向直接后继 P3 发送信号,表示进程 P2 执行完毕
}
P3()
{
    P(a1);                      //等待直接前趋 P1 发送已完成的信号
    P(a2);                      //等待直接前趋 P2 发送已完成的信号
    c = a + b;                  //执行进程的主体代码
    V(a3);                      //向直接后继 P4 发送信号,表示进程 P3 执行完毕
}
P4()
{
    P(a3);                      //等待直接前趋 P3 发送已完成的信号
    d = c + b;                  //执行进程的主体代码
}
```

3.5 经典的进程同步问题

在多道程序环境下，进程同步问题十分重要，也是相当有趣的问题，因而引发了不少学者对它进行研究，由此产生了一系列经典的进程同步问题，其中较有代表性的是“生产者-消费者问题”“读者-写者问题”“哲学家进餐问题”等。通过对这些问题的学习和研究，可以帮助我们更好地理解进程同步概念及实现方法。

3.5.1 生产者-消费者问题

例3-8 生产者-消费者（producer-consumer）问题是一个著名的进程同步问题。它描述的是：有一群生产者进程（producer(i)）在生产产品，并将这些产品提供给消费者进程

(consumer(*i*))去消费。为使生产者进程与消费者进程能并发执行,在两者之间设置了一个具有 *n* 个缓冲区的环形缓冲池,如图 3-6 所示。生产者进程将它所生产的产品放入一个个缓冲区中;消费者进程可从一个个缓冲区中取走产品去消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的,但它们之间必须保持同步,即不允许消费者进程到一个空缓冲区中取产品,也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

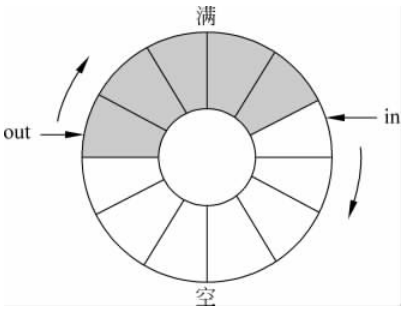

图 3-6 环形缓冲池

1. 利用信号量机制来解决生产者-消费者问题

可利用一个数组来表示上述具有 *n* 个(0,1,…,*n*−1)缓冲区的缓冲池。用输入指针 in 来指示下一个可投放产品的缓冲区,每当生产者进程生产并投放一个产品后,输入指针加 1。用输出指针 out 来指示下一个可从中获取产品的缓冲区,每当消费者进程取走一个产品后,输出指针加 1。由于这里的缓冲池是循环缓冲的,故应把输入指针加 1 表示成 in=(in+1)%*n*,输出指针加 1 表示成 out=(out+1)%*n*。当(in+1)%*n*=out 时表示缓冲池满,而 in=out 则表示缓冲池空。

可利用互斥信号量 mutex 实现各进程对缓冲池的互斥使用;利用信号量 empty 和 full 分别表示缓冲池中空缓冲区和满缓冲区的数量。对生产者-消费者问题的算法描述如下:

```
int   in = 0,   out = 0;                    //定义和初始化全局变量
semp   empty = n;                           //设置信号量 empty,初值为 n 表示空缓冲区的数量
semp   full = 0;                            //设置信号量 full,初值为 0 表示满缓冲区的数量
semp mutex;                                 //用于缓冲区的互斥信号量
var   buf[n];                               //定义一个全局缓冲区
producer(i)                                 //生产者进程向缓冲区送数
{
    var nextp;                              //定义局部变量用于存放每次刚生产出来的产品
    while (true)
    {
        生成新的产品放入 nextp;
        P(empty);
        P(mutex);
        Buffer[ in] = nextp;
        in = ( in + 1) % n;
        V(mutex);
        V(full);
    }
}
consumer(i)                                 //消费者从缓冲区中取走产品
{
    var nextc;                              //定义局部变量用于存放每次要消费的产品
    while (true)
    {
        P(full);
```

```
        P(mutex);
        nextc = buffer[out];
        out = (out + 1) % n;
        V(mutex);
        V(empty);
        处理产品 nextc;
    }
}
```

注意：在生产者-消费者问题中，由于同一过程中包含几个信号量，因此，对P、V原语的操作次序要非常小心。一般来说，由于V原语是释放资源的，所以可以以任意次序出现。但P原语则不然，如果次序混乱，将会造成进程之间的死锁。

2. 利用管程实现生产者-消费者问题

```
Monitor  ProducerConsumer {
    integer: count, in, out;          //数据结构定义
    buf: array [0..n - 1] of item_type;
    condition: full, empty;
procedure put(item)                   //过程
{
    if count > = n then empty.wait;
    buf[in] = item;
    in: = (in + 1) mod n;
    count++;
    if full.queue then full.signal;
}
procedure get(item)
{
    if count < = 0 then full.wait;
    item = buf[out];
    out: = (out + 1) mod n;
    count --;
    if empty.quence then empty.signal;
}
{  in = 0;  out = 0;  count = 0;  }   //初始值
producer()                            //生产者进程
{
    while (true)
    {
        produce(item);
        ProducerConsumer.put(item);
    }
}
consumer()                            //消费者进程
{
    while (true)
```

```
    {
        ProducerConsumer.get(item);
        consume(item);
    }
}
```

通过临界区互斥的自动化,管程比信号量更能保证并发编程的正确性,但编译器必须识别管程并使用某种方法保证管程的互斥执行。

生产者-消费者问题是相互合作的进程关系的一种抽象,我们把系统中使用某一类资源的进程称为该类资源的消费者,而把释放同类资源的进程称为该类资源的生产者。例如,在例 3-4 中,采集进程是生产者,计算进程是消费者。因此,生产者-消费者问题具有很大的代表性和实用价值。

3.5.2 读者-写者问题

例 3-9 一个数据文件或记录可被多个进程共享,把只要求读该文件的进程称为读者(Reader)进程,其他进程则称为写者(Writer)进程,如图 3-7 所示。允许多个进程同时读一个共享对象,因为读操作不会使数据文件混乱。但不允许一个写者进程和其他读者进程或写者进程同时访问共享对象,因为这种访问将会引起混乱。所谓读者-写者问题(Reader-Writer Problem)是指保证一个写者进程必须与其他进程互斥地访问共享对象的同步问题。

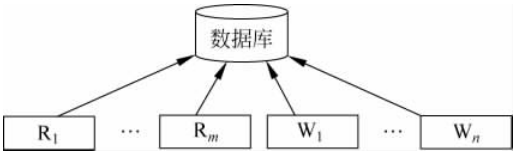

图 3-7 读者-写者问题

为实现读者与写者进程间在读或写时的互斥而设置了一个互斥信号量 Wmutex。另外,再设置一个整型变量 Readcount 表示正在读的进程数目。由于只要有一个读者进程在读,便不允许写者进程去写。因此,仅当 Readcount=0,表示尚无读者进程在读时,读者进程才需要执行 P(Wmutex)操作。若 P(Wmutex)操作成功,读者进程便可去读,相应地,做 Readcount+1 操作。同理,仅当读者进程在执行了 Readcount-1 操作后其值为 0 时,才须执行 V(Wmutex)操作,以便让写者进程写。又因为 Readcount 是一个可被多个读者进程访问的临界资源,因此,应该为它设置一个互斥信号量 Rmutex。其算法描述如下:

```
int  Readcount = 0;                    //读者进程的数目
semp  Wmutex = 1;                      //读者与写者的互斥信号量
semp  Rmutex = 1;                      //Readcount 的互斥信号量

Reader(i)
{
    while (true)
    {
        P(Rmutex);
        if(Readcount == 0)  P(Wmutex); //第一个进来的读者
```

```
        Readcount++;
        V(Rmutex);
        读数据库;
        P(Rmutex);
        Readcount--;
        if(Readcount == 0)  V(Wmutex); //最后一个离开的读者
        V(Rmutex);
    }
}
Writer (i)
{
    while (true)
    {
        P(Wmutex);
        写数据库;
        V(Wmutex);
    }
}
```

3.5.3 哲学家进餐问题

例 3-10 哲学家进餐问题也是一个经典的同步问题。该问题的描述如下：5 位哲学家围坐在一张圆桌周围，桌子中间放了一盘食品，相邻两位哲学家之间有一只筷子，如图 3-8 所示。哲学家的生活包括两种活动，即吃饭和思考(这只是一种抽象，即对本问题而言其他活动都无关紧要)。当一位哲学家觉得饿时，他试图分两次取其左右最靠近他的筷子，每次拿一只，但不分次序。如果成功获得两只筷子，他就开始吃饭，吃完以后放下筷子继续思考。为每一个哲学家写一段程序来描述其行为。

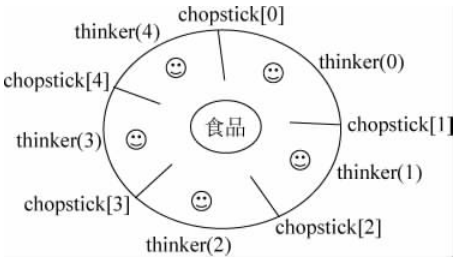

图 3-8 哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这 5 个信号量构成信号量数组。第 i 位哲学家的行为算法描述如下：

```
semp  chopstick[4] = {1, 1, 1, 1, 1};   //筷子的互斥信号量
thinker(i)
{
    P(chopstick[i]);
    P(chopstick[(i + 1) mod 5]);
    进餐;
```

```
    V(chopstick[i]);
    V(chopstick[(i+1) mod 5]);
    思考;
}
```

在以上描述中,当哲学家饥饿时,总是先去拿他右边的筷子,即执行 P(chopstick[i]);成功后,再去拿他左边的筷子,即执行 P(chopstick[(*i*+1) mod 5]),又成功后便可进餐。进餐毕,他先放下右边的筷子,然后再放左边的筷子。虽然,上述解法可保证不会有两个相邻的哲学家同时进餐,但有可能引起死锁。假如 5 位哲学家同时饥饿而各自拿起右边的筷子时,就会使 5 个信号量 chopstick 均为 0;当他们再试图去拿左边的筷子时,都将因无筷子可拿而无限期地等待。对于这样的死锁问题,可采取以下几种解决方法:

(1) 至多只允许有 4 位哲学家同时去拿右边的筷子,最终能保证至少有一位哲学家能够进餐,并在用毕时能释放出他用过的两只筷子,从而使更多的哲学家能够进餐。

(2) 规定偶数号哲学家先拿他右边的筷子,再拿左边的筷子;而奇数号哲学家则相反。按此规定,将是 1、2 号哲学家竞争 2 号筷子,3、4 号哲学家竞争 4 号筷子。即,5 位哲学家都先竞争偶数号筷子,获得后,再去竞争奇数号筷子,最后总会有一位哲学家能获得两只筷子而进餐(读者可自己写出其算法)。

信号量机制是一种有效的进程同步工具。在长期而广泛的应用中,信号量机制又得到了很大的发展。例如,对上述问题除了利用前面介绍的整型信号量机制解决以外,也可采用记录型信号量或信号量集等机制解决。有兴趣的读者可进一步查阅相关的资料。

3.6 进程通信

并发执行的进程为了协调一致地完成指定的任务,进程之间要有一定的联系,这种联系通常采用进程间交换数据(或信息)的方式进行,我们将这种方式称为进程的通信。

3.6.1 进程通信的类型

进程通信交换的数据量可多可少,在操作系统中将数据交换量少的进程协调过程称为低级通信,而将交换信息量较大的过程称为高级通信(也称消息通信)。

低级通信由于数据量小,通常交换的是控制信息,一般传递一个或几个字节的信息,有时仅仅为一个状态、标志或数值,它们常采用变量、数组等方式实现。进程间的互斥与同步,由于其所交换的信息量少而被归入低级通信,进程通过修改信号量来向其他进程表明该资源是否可用。应当指出,信号量机制作为同步工具是有效的,但作为通信工具却不够理想,这是因为共享数据结构的设置、数据的传送、进程的互斥与同步都必须由程序员实现。这不仅增加了程序设计的复杂性,也给程序理解带来困难,且 P、V 操作易导致死锁。

高级通信由于交换的信息数据量大,进程间可采用缓冲、信箱、管道和共享区等方式实现。这种大量的传递促进了本地进程间的通信和远程进程间的通信的开发,从而为远程终端操作和计算机网络的开发和控制奠定了基础。本节重点讨论高级通信。

3.6.2 进程通信的方式

根据通信实施的方式和数据存取的方式，进程通信方式可归结为共享存储器方式、消息缓冲方式和管道通信方式。随着网络的发展，进程间通信出现了套接字、远程调用等方法，相关内容可以参考计算机网络教材。

1. 共享存储器

共享存储器方式的通信基础是共享数据结构或共享存储区，进程之间能够通过这些空间进行通信。数据结构是系统为保证进程正常运行而设置的专门机制，利用某个专门的数据结构存放进程间需交换的数据，它可以指定为一个寄存器、一组寄存器、一个数组、一个链表、一个记录等。例如，可以在每个进程的 PCB 表中增加一个表项来存放通信信息，进程由通信表项中取得交换数据。共享存储区是在主存中设置一个专门的区域，进程像生产者和消费者一样共用这个存储区送数和取数。这里，公用数据结构的设置及对进程间同步的处理都是程序员的职责，这无疑增加了程序员的负担，而操作系统却只须提供共享存储器。因此，这种通信方式是低效的，只适于传递少量的数据。

2. 消息缓冲

不论是单机系统、多机系统还是计算机网络，消息缓冲机制都是应用最广泛的一种进程间通信的机制。在消息缓冲系统中，进程间的数据交换是以格式化的消息(message)为单位的；在计算机网络中，又把 message 称为报文。程序员直接利用系统提供的一组通信命令进行通信。操作系统隐藏了通信的实现细节，大大降低了通信程序编制的复杂性，因而使消息缓冲方式获得广泛的应用。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

3. 管道通信

所谓管道，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名管道文件。向管道(共享文件)提供输入的发送进程(即写进程)以字符流形式将大量的数据送入管道，而接收管道输出的接收进程(即读进程)则从管道中接收(即读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于 UNIX 系统，由于它能有效地传送大量数据，因而又被引入到许多其他操作系统中。

为了协调双方的通信，管道机制必须提供以下 3 方面的协调能力：

(1) 互斥，即当一个进程正在对管道执行读/写操作时，其他(另一)进程必须等待。

(2) 同步，指当写(输入)进程把一定数量(如 4KB)的数据写入管道，便去睡眠等待，直到读(输出)进程取走数据后，再把它唤醒。当读进程读一个空管道时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

(3) 确定对方是否存在，只有确定了对方已存在时，才能进行通信。

3.6.3 消息缓冲队列通信机制

消息缓冲队列通信机制首先由美国的 Hansan 提出，并在 RC 4000 系统上实现，后来被

广泛应用于本地进程之间的通信中。操作系统将一组数据称为一个消息,并在系统中设立一个大的缓冲区,作为消息缓冲池,缓冲池分成一个个的消息缓冲区,每个缓冲区中存放一个消息。进程通信时,首先向系统申请一个缓冲区,放入自己的消息,并通知接收进程。接收进程从缓冲区中取走消息,同时释放缓冲区交回系统。消息通信通常采用一对系统调用,即 Send(发送)过程和 Receive(接收)过程来实现。

1. 消息缓冲队列通信机制中的数据结构

1) 消息缓冲区

在消息缓冲队列通信方式中,主要利用的数据结构是消息缓冲区,它是一个记录结构,主要包含下列内容:

```
sender;                          //发送者进程标识符
size;                            //消息长度
text;                            //消息正文
next;                            //指向下一个消息缓冲区的指针
```

2) PCB 中有关通信的数据项

在利用消息缓冲队列通信机制时,在设置消息缓冲队列的同时,还应增加用于对消息队列进行操作和实现同步的信号量,并将它们置入进程的 PCB 中。在 PCB 中应增加的数据项可描述如下:

```
mq;                              //消息队列队首指针
mutex;                           //消息队列互斥信号量,初值为 1
sm;                              //消息队列资源信号量,用于进程的消息计数,初值为 0
```

2. 发送过程

发送进程在调用发送过程发送消息之前,应先在自己的内存空间设置一个发送区 a,如图 3-9 所示,把待发送的消息正文、发送进程标识符、消息长度等信息填入其中,然后调用发送过程,把消息发送给目标(接收)进程。发送过程首先根据发送区 a 中所设置的消息长度 a.size 从缓冲池中申请一个缓冲区 i,接着,把发送区 a 中的信息复制到缓冲区 i 中。为了能将 i 挂在接收进程的消息队列 mq 上,应先获得接收进程内部标识符 j,然后将 i 挂在 j.mq 上。由于该队列属于临界资源,故在执行 insert 操作的前后都要执行 P、V 操作。

发送过程可描述如下:

```
send(receiver, a)
{
    getbuf(a.size,i);            //根据 a.size 申请缓冲区
    i.sender = a.sender;         //将发送区 a 中的信息复制到消息缓冲区中
    i.size = a.size;
    i.text = a.text;
    i.next = 0;
    getid(PCB set,receiver,j)    //获得接收进程内部标识符 j
    P(j.mutex);
    insert(j.mp,i);              //将消息缓冲区插入消息队列
    V(j.mutex);
    V(j.sm);                     //通知接收进程
}
```

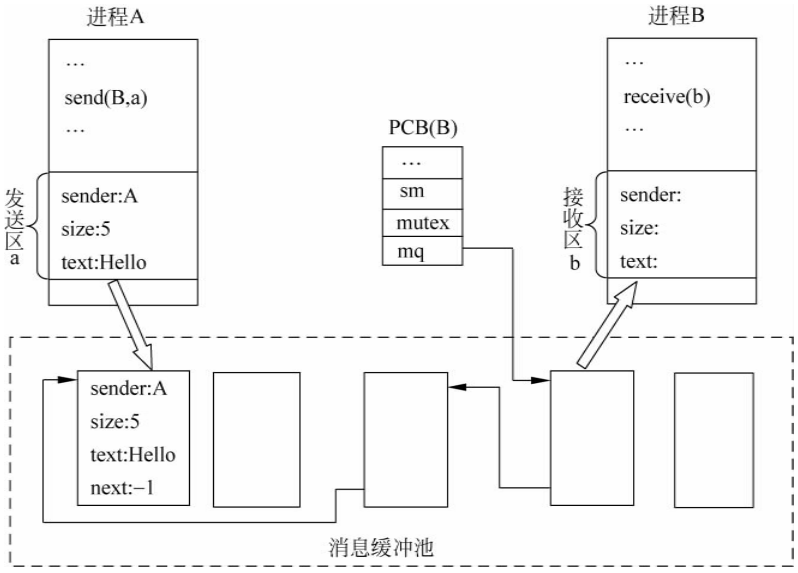

图 3-9 采用消息缓冲队列通信机制的消息接收和发送过程

3. 接收过程

接收进程调用接收过程 receive(b)从自己的消息缓冲队列 mq 中摘下第一个消息缓冲区 i,并将其中的数据复制到以 b 为首址的指定消息接收区。接收过程描述如下:

```
receive(b)
{
    j = getid();                    //获得接收进程内部标识符 j
    P(j.sm);
    P(j.mutex);
    remove(j.mq,i);                 //将消息队列中的第一个消息移出
    V(j.mutex);
    b.sender = i.sender;            //将消息缓冲区 i 中的信息复制到接收区 b
    b.size = i.size;
    b.text = i.text;
    releasebuf(i);                  //将消息缓冲区 i 释放
}
```

3.6.4 信箱通信

进程通信也可以采用信箱通信(mailbox)的方式。信箱是一种大小固定的私有数据结构,它不像缓冲区那样被系统内所有进程共享。它由信箱头和若干信箱体组成。其中,信箱头描述信箱名称、大小、方向以及拥有该信箱的进程名等,信箱体主要用来存放消息。图 3-10 为信箱通信结构。

当进程 A 希望与进程 B 通信时,由进程 A 创建一个连接两个进程的信箱。在以后的通信中,进程 A 将调用发送过程将信件投入信箱,系统保证进程 B 可在任何时候调用接收过程取走信件而不丢失。因此,利用信箱通信方式,既可以实现实时通信,也可以实现非实时通信。

图 3-10 信箱通信结构

3.7 线程

20 世纪 80 年代中期,人们提出了比进程更小的能独立运行的基本单位——线程,试图用它来提高系统内并发执行的程度,进一步提高系统的吞吐量。

3.7.1 线程的引入

许多实际(应用)系统,如事务处理软件、数据库处理软件、窗口系统以及操作系统本身等,经常需要同时处理多个服务请求,而且对这些服务请求的处理不仅运行的是同一服务程序,更是针对同一地址空间(同一数据区)的。例如,航空公司售票系统需要同时处理来自多个售票窗口的购票或查询请求,对这些购票或查询请求的处理都是对同样的数据——"飞机座位和售出信息"进行的(可能针对同一航班或不同航班)。又如,数据库服务器软件需要同时处理来自多个客户机的数据查询请求,这些请求都是针对同一数据库的。再如,操作系统需要同时处理来自多个用户进程的读盘请求,这些请求都针对同一个盘,对这些请求的处理都是基于同一磁盘缓冲区。对于以上这种"基于同一数据区同时多个请求"的情况,用进程模型来实现时,显然只有 3 种办法:

(1) 用一个进程顺序处理所有请求,当该进程正在处理一个购票请求时(即使该进程正在因为处理该请求而处于等待磁盘服务期间,也就是说,不管该进程是在忙时还是在等时),其他购票请求只能等待。例如,在航空公司售票系统中,用一个进程来处理来自所有售票窗口的所有购票请求。显然,这种方案会导致较多的等待(较长的等待队列)和较慢的响应时间。其中关键的效率问题是,当该进程因处理当前请求而需要等待磁盘服务(或其他资源)时,即使还有其他请求要处理,该进程也进入等待态,这样就出现了一方面有很多请求等待处理,另一方面该服务进程却处于等待态的矛盾和时间浪费局面。

(2) 用多个相互独立的进程,每个进程负责处理一个购票请求。这种方案不会出现上述的矛盾和时间浪费局面,但显然,这些进程间需要大量的和复杂的共享机制,而且需要大量的进程,每个进程都需要占用一套完整的进程管理信息,这些进程频繁地动态建立和撤销,频繁地进行进程切换。这些开销是很大的,而考虑到这些进程处理的大部分数据是共享的,运行的程序也是同一个程序,这种开销就更值得研究了。

(3) 用一个进程来并发处理所有请求,只要还有其他请求要处理,该进程就不进入等待态。例如,当该进程因处理当前请求而需要等待磁盘服务时,该进程记录当前请求的当前处理状态,然后转去处理下一请求;而当前一请求所等待的磁盘服务完成时,该进程需在适当的时间继续为前一请求服务。显然,在这种方案下,该进程需要记录所有请求的处理状态,并在这些请求间进行切换和轮换服务(整个进程的操作类似于一个有限状态机:根据发生

的事件作出相应的反应)。这加重了该进程的负担和复杂性。而实际上,这种管理负担是一种与应用无关的共性的需要,不应由每个用户进程来承担,而应考虑由操作系统和一个公用函数库来统一实现。

从上述分析可以看出,以上 3 种办法都不能很好地解决和实现“基于同一数据区同时处理多个请求”的需要。因而,有不少研究操作系统的学者想到,若能将进程作为拥有资源的基本单位,不作为调度的基本单位,不对之进行频繁的切换,从而减少程序并发执行时系统所付出的时空开销,就能够提高进程执行的并发程度,提高资源的利用率和系统的吞吐量。正是在这种思想的指导下,形成了线程的概念。

3.7.2 线程的概念

一个进程内的基本调度单位称为线程或轻权进程,这个调度单位既可由操作系统的内核控制,也可由用户程序控制。在引入多线程的操作系统中,进程和线程具有如下的区别和联系:

(1) 进程作为系统资源分配的基本单位,与进程有关的资源信息都被记录在进程控制块 PCB 中,以表示该进程拥有这些资源或正在使用它们。在任一进程中所拥有的资源包括受到保护的进程地址空间、用于实现进程间和线程间同步和通信的机制、已打开的文件和已申请到的 I/O 设备,以及一张由系统核心维护的地址映射表,该表用于实现用户的程序的逻辑地址到其物理地址的映射。线程中的实体基本上不拥有系统资源,只有一点必不可少的、能保证独立运行的资源。例如,在每个线程中都应具有一个用于控制线程运行的线程控制块(Thread Control Block,TCB),用于指示被执行指令的程序计数器,以及保留局部变量、少数状态参数和返回地址的一组寄存器和堆栈。

(2) 进程不再是一个独立运行的基本单位,而是将线程作为一个独立运行的基本单位。由于线程很“轻”,相对于进程切换,线程的切换非常迅速且开销小。而且进程的调度与切换都是由操作系统内核完成的,而线程既可由操作系统内核完成,也可由用户程序完成。

(3) 一个系统中可以有多个进程,一个进程可以有一个或多个线程(至少有一个)。系统中的所有线程都只能属于某一特定的进程。这些线程都能并发执行,但只有在多处理机系统中它们才能真正地并行运行。

(4) 在同一进程中的各个线程都可以共享该进程所拥有的资源。由于同一进程的所有线程都具有相同的地址空间(同一进程的地址空间),所以,每个线程都可访问每个虚地址,一个线程可以读、写甚至完全破坏另一个线程的堆栈。线程之间没有保护,因为不可能也不必要。不同的进程可能来自不同的可能相互敌对的用户,而某个进程总是由一个特定的用户所有,它创建多个线程只是为了协作,而不是为了冲突。除了共享地址空间外,所有的线程还共享同一组打开的文件、子进程、定时器、信号等。

(5) 与传统的进程类似,线程可以创建子线程,在各线程之间也存在着共享资源和相互合作的制约关系,致使线程运行时也具有 3 种基本的状态:运行、阻塞、就绪。

多线程系统中,进程与线程的关系如图 3-11 所示。

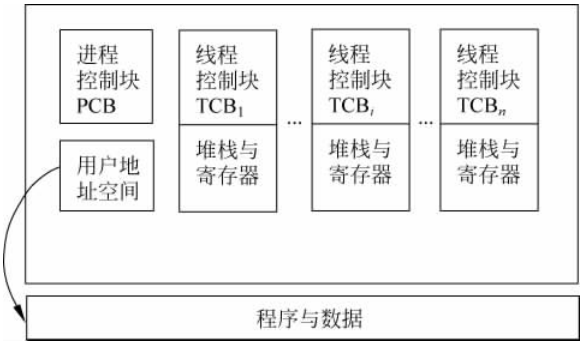

图 3-11 线程与进程的关系

3.7.3 线程的控制

线程的控制通常以线程包的形式体现,一组供用户使用的与线程有关的原语(即系统调用)称为一个线程包。

1. 线程的建立与撤销

在多线程操作系统环境下,应用程序在启动时通常仅有一个线程在执行,该线程称为初始化线程。它可根据需要再去创建若干个线程。在创建新线程时要利用线程创建函数(或系统调用),并提供相应的参数,如指向线程主程序的入口指针、堆栈的大小以及用于调度的优先级等。在线程创建函数执行完后,将返回一个线程标识符供以后使用。

与进程类似,线程可有两种终止方式:因完成任务而自行终止,或被外界强行终止。但有些线程(主要是系统线程)一旦建立便一直运行下去而不再被终止。

2. 线程调度

线程调度算法和进程调度算法是类似的,例如优先级、轮转、多重队列调度算法等。和进程模型一样,线程包也提供相应的界面,允许用户选择调度算法和设置优先级。

3. 进程全局变量和线程(私有)全局变量

线程不仅有自己的栈和程序计数器,有时还需要有少量的私有数据。这样,在多线程系统中,一个进程中的所有变量便分为 3 类:进程全局变量(对该进程中的所有线程中的所有过程可见)、线程(私有)全局变量(对该线程中的所有过程可见)、过程局部变量(只对该过程可见)。由于目前的程序设计语言只支持进程全局变量和过程局部变量,更确切地说,目前的语言只支持全局变量(对整个程序而言)和局部变量(对一个过程而言),而并不对全局变量作更细的划分,因此,对线程(私有)全局变量,需要将线程(私有)全局变量数据区的地址作为一个额外的参数传递给该线程中的每个过程。

4. 线程互斥与同步

由于同一进程内的各线程共享该进程的所有资源和地址空间,任何线程对资源的操作

都会对其他相关线程带来影响。因此，系统必须为线程的执行提供同步控制机构。线程中所使用的同步控制机构与进程中所使用的相同。

3.7.4 线程的实现

对于线程管理有两种方式：一种是由操作系统来管理线程，另一种是由进程自己来管理线程。因此，线程的实现就有两种方式：用户态线程和核心态线程。在同一个操作系统中，有的使用纯用户态线程，有的使用纯核心态线程，有的则混合使用这两种方式。

1. 用户态线程

这种方式将线程包完全放在用户空间内，而核心对此一无所知。就核心而言，它只是在管理常规的进程——即单线程进程。线程在一个线程运行管理系统上执行，而线程运行管理系统则是一组管理线程的过程。当线程执行系统调用、转入睡眠、实施一个信号量或互斥量操作或其他可能导致它被挂起的操作时，都调用线程运行管理系统中的过程，这个过程检查线程是否必须被阻塞。若是，它将线程的寄存器存入表中，寻找一个未被阻塞的线程来运行，并为新线程装配寄存器。一旦堆栈指针和程序计数器被切换，新线程就立即被激活了。如果计算机有存储所有寄存器及恢复所有寄存器的指令，则整个线程切换工作用几条指令就可以完成。即，线程在切换时只进行线程执行环境的切换，不进行处理机的切换。

2. 核心态线程

这种方式下，用户级的线程运行管理系统已不再需要了。核心为每个进程准备了一张表，每个线程占一项，填写有关的寄存器、状态、优先级和其他信息。这些信息与用户态线程是一样的，只是现在放在核心空间。所有对线程的操作都以系统调用的形式实现。当线程被阻塞时，操作系统不仅可以运行同一进程中的另一线程，而且可以运行别的进程中的线程。相比之下，用户级的线程运行管理系统总是在运行本进程中的线程，除非操作系统核心取消其 CPU 使用权(或已经没有就绪的线程供运行了)。

3. 对用户态线程和核心态线程的评价

1) 用户态线程的优点和核心态线程的缺点

(1) 用户态线程最明显的优点是它可以在一个不支持线程的操作系统上实现。例如，UNIX 并不支持线程，但已有了多个基于 UNIX 的用户态线程包。

(2) 开销和性能。用户态线程切换比陷入核心至少快一个数量级，这也就是用户态线程的受欢迎之处。而核心态线程中，由于所有线程操作都以系统调用的形式实现，从而比在用户级调用线程运行管理系统中的过程开销大得多。

(3) 用户态线程允许每个进程有自己特设的调度算法。对有些应用，如配有一个空闲区回收线程的应用，有了自己的调度算法，就不必担忧一个线程会在一些不适当的地方停下来。

(4) 用户态线程的可扩充性也很好。而核心线程需要不停地使用核心空间，这在线程数较多时是一个问题。

2) 用户态线程的缺点和核心态线程的优点

用户态线程虽然有较好的性能,但也有一些较大的问题。

(1) 在用户态线程中,阻塞型系统调用会阻塞所有的线程。例如,线程在读一条空的管道时,会导致所在进程阻塞,这意味着该进程的所有线程(包括本线程)都被阻塞。而在核心实现中,同样情况发生时线程陷入内核,内核将线程挂起,并开始运行另一个线程。

(2) 在用户态线程中,在一个线程开始运行以后,除非它自愿放弃 CPU,否则没有其他线程能得到运行。而在核心级线程中,周期发生的时钟中断可以解决这个问题。在用户态线程实现中,单进程中没有时钟中断,从而轮转式的调度是行不通的。

3) 用户态线程和核心态线程都存在的问题

典型的问题是,由于一个进程内的所有线程共享该进程的所有数据区和信号等资源,因此随机的线程切换会导致数据的覆盖、不一致等错误,导致资源使用的冲突,导致很多库程序变成不可再入代码。

3.7.5 线程的适用范围

使用线程最大的好处是在有多个任务需要处理机处理时能减少处理机的切换时间。线程的几种典型的应用如下:

(1) 服务器中的文件管理或通信控制。局域网的文件服务器(进程)由于等待盘操作而经常被阻塞。若对文件的访问要求由服务器进程派生出的线程进行处理,则第一个线程睡眠时另一个线程可以继续运行(接受新的文件服务请求)。这样的好处是文件服务的吞吐率和性能都提高了。如果计算机系统是多处理机的,这些线程还可以被安排到不同的处理机上执行。

(2) 线程也经常用于客户线程。例如,一个客户想要在多个服务器上备份一个文件,就可以用一个线程与一个服务器对话。客户线程的另一种用途是处理信号,如 Del、Break 等键盘中断。这时,不再让信号中断进程,而是专门安排一个线程来等待中断。通常它是被阻塞的,当信号产生时,它被唤醒并处理该信号。因此,用了线程后可以消除用户级的中断。

(3) 前后台处理。许多用户都有过前后台处理经验,即把一个计算量较大的程序或实时性要求不高的程序安排在处理机空闲时执行。对于同一个进程中的上述程序来说,线程可用来减少处理机切换时间和提高执行速度,例如,在表处理进程中,一个线程可用来显示菜单和读取用户输入,而另一个线程则可用来执行用户命令和修改表格。由于用户输入命令和命令执行分别由不同的线程在前后台执行,从而提高了操作系统的效率。

(4) 数据的批处理以及网络系统中的信息发送与接收和其他相关处理等。例如,图 3-12 给出了一个用户主机通过网络向两台远程服务器进行远程调用(RPC)以获得相应结果的执行情况。如果用户程序只用一个线程,则第二个远程调用的请求只有在得到第一个请求的执行结果后才能发出,如图 3-12(a)所示。采用多线程时,用户程序不必等待第一个 RPC 请求的执行结果而直接发出第二个 RPC 请求,如图 3-12(b)所示,从而缩短了等待时间。

由此,我们知道,最适合线程的系统是多处理机系统。但是,并不是所有的计算机系统都适合使用线程。事实上,在那些很少做进程调度和切换的实时系统、个人数字助理系统中,由于任务的单一性,设置线程反而会占用更多的内存空间和寄存器。同时,使用线程不当可能导致死锁。

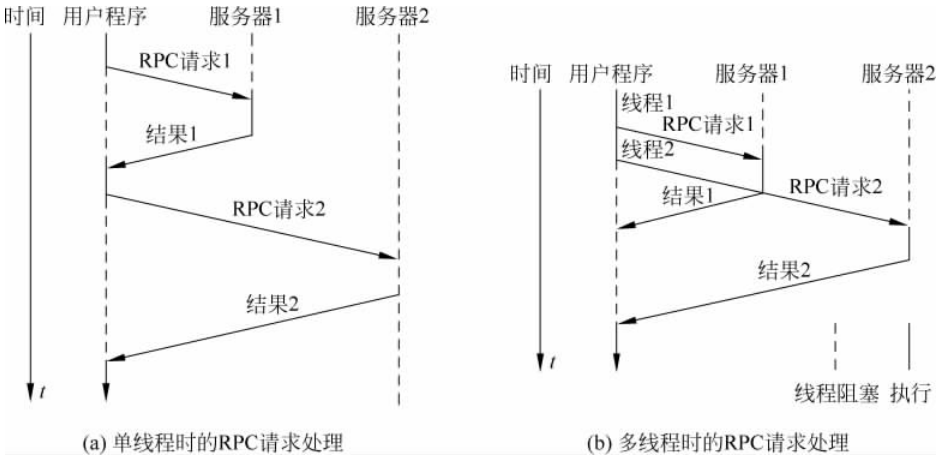

图 3-12 RPC 请求处理

3.8 Linux 的进程管理

进程是 Linux 系统中一个重要的概念，Linux 系统的一个重要特点就是可以同时启动多个进程。本节主要介绍 Linux 进程描述、进程的状态、进程的控制及进程通信。

3.8.1 Linux 进程概念与描述

Linux 进程符合一般操作系统教材中对进程概念的解释，即进程是一个程序的一次执行的过程，进程是系统中最基本的执行单位。程序是静态的，它是一些保存在磁盘上的可执行的代码和数据集合；而进程是一个动态的概念。

在 Linux 系统中，进程仍是最小的调度单位。进程被存放在叫做任务链表(tasklist)的双向循环链表中。Linux 中的进程控制块是一个名叫 task_struct 的数据结构，其中包含了很多重要的信息，供系统调度和进程本身执行使用。

每个进程用一个 task_struct 数据结构来表示(任务与进程在 Linux 中可以混用)。数组 task 包含指向系统中所有 task_struct 结构的指针。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。task_struct 结构部分描述如下：

```
struct task_struct {
    volatile long state;          /* 进程的状态 */
    unsigned long flags;          /* 进程标志 */
    mm_segment_t addr_limit;      /* 线性地址空间 */
    …
    long counter;                 /* 进程的动态优先级 */
    long nice;                    /* 进程的静态优先级 */
    unsigned long policy;         /* 进程采用的调度策略 */
    struct mm_struct * mm;        /* 进程属性中指向内存管理的数据结构 mm_structd 的指针 */
    …
```

```
    struct task_struct *next_task, *prev_task;    /*进程通过这两个指针组成一个双向链表*/
    struct mm_struct *active_mm; /*指向活动地址空间*/
    /*task state*/
    pid_t pid;                    /*进程标识符*/
    gid_t gid;                    /*进程组标号*/
    uid_t uid                     /*用户标识*/
    …
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
    /*这5个标志表示一个进程在计算机中的亲属关系,分别代表祖先进程、父进程、子进程、弟进
    /*程和兄进程,为了在两个进程之间共享方便而设立*/
    struct list_head thread_group;
    …
    struct task_struct *pidhist_next;
    struct task_struct *pidhist_pprev;
                                  /*上面两个指针是为了在计算机中快速查一个进程而设立的*/
    …
    struct fs_struct *fs;         /*指向和文件管理有关的数据结构*/
    …
};
```

task_struct 中最重要的信息为进程 ID 在上述结构中定义为 pid,进程 ID 也被称作进程标识符(Process ID,PID),是一个非负的整数,在 Linux 操作系统中唯一地标识一个进程。

Linux 中的进程分为普通进程和实时进程两种,实时进程必须对外部事件作出快速反应,实时进程的优先级高于普通进程。

3.8.2 Linux 中的进程状态及其转换

Linux 中的进程有 5 种状态:

(1) 可运行状态(TASK_RUNNING)。相当于进程 3 种基本状态中的执行状态和就绪状态,正在运行或准备运行的进程处于这种状态,处于这种状态的进程实际参与进程的调度。

(2) 可中断阻塞状态(TASK_INTERRUPTIBLE)。处于这种阻塞状态中的进程,通常只要阻塞的原因解除,比如请求资源未能满足而阻塞,一旦资源满足后,就可以被唤醒到就绪状态;也可以由其他进程通过信号或定时中断唤醒,并进入就绪队列。这种阻塞状态类似于一般进程的阻塞状态。

(3) 不可中断阻塞状态(TASK_UNINTERRUPTIBLE)。处于这种阻塞状态的进程,只能在资源请求得到满足时唤醒到就绪状态,不能通过信号或定时中断唤醒。

(4) 僵死状态(TASK_ZOMBIE)。处于这种状态的进程已经结束运行,离开 CPU,并归还所占用的资源,只是进程控制块(PCB)结构还没有归还释放。

(5) 暂停状态(TASK_STOPPED)。处于这种状态的进程被暂停执行而阻塞,通过其他进程的信号才能唤醒。导致暂停的原因有两种:一是收到暂停信号(SIGSTOP、SIGSTP、SIGTTIN、SIGTTOU);二是受其他进程的系统调用的控制,而暂时把 CPU 交给控制进程,处于暂停状态。

这 5 种状态不是固定不变的,它们随着条件的变化而转换,如图 3-13 所示。

用户进程执行 do_fork()函数时创建一个新的子进程,该子进程插入就绪队列,处于可

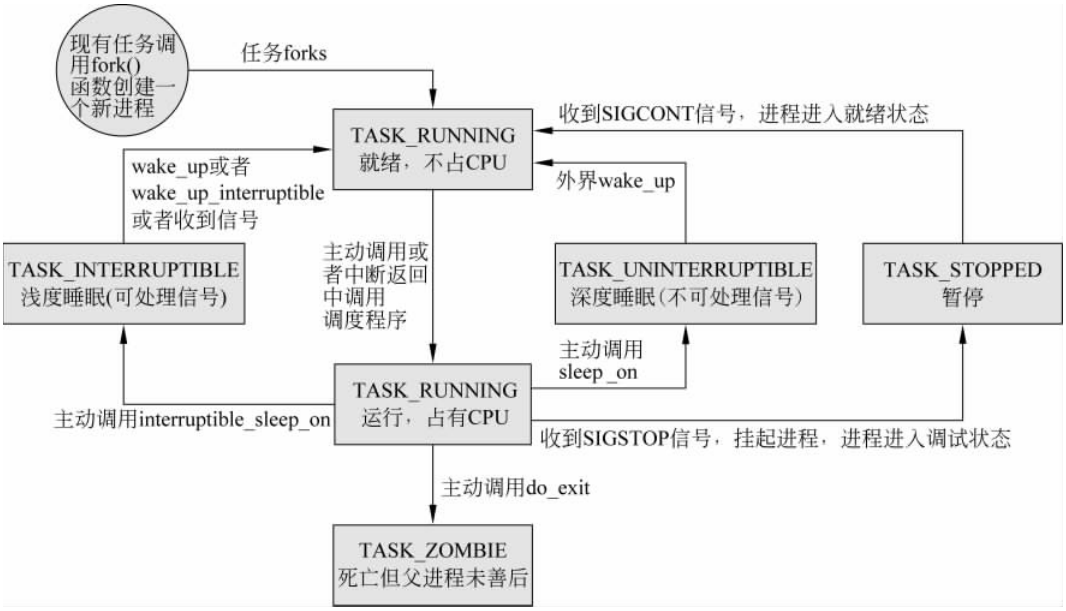

图 3-13 Linux 进程状态转换

运行状态。创建一个子进程时，进程状态为不可中断阻塞状态。在创建子进程的工作结束前把父进程唤醒为就绪状态，即可运行状态。处于可运行状态的进程插入就绪队列中，在适当的时候被调度程序选中，可以获得 CPU。占有 CPU 的进程，当分给它的时间片(10ms 的整数倍)用完时，由时钟中断触发重新调度，使该进程又回到就绪状态，并挂到就绪队列队尾。

已经占有 CPU 并正在运行的进程，若申请资源不能满足，则睡眠阻塞。若调用 sleep_on()，睡眠状态变为不可中断阻塞状态。若调用 interrupt_sleep_on()，睡眠状态变为可中断阻塞状态。一旦进程变为阻塞状态，其释放的 CPU 会马上被调度程序重新调度一个就绪进程去占用，而阻塞的进程插入相应的等待队列，一旦资源满足后，阻塞的进程就可以被唤醒到就绪状态；也可以由其他进程通过信号或定时中断唤醒，变为运行状态。而处于不可中断阻塞状态的进程只能由所请求的资源得到满足而唤醒，不能由信号或定时中断唤醒。唤醒后插入就绪队列。

当进程执行系统调用 sys_exit()或收到 SIG KILL 信号(取消进程)而调用 do_exit()结束时，进程变为僵死状态。此时，归还它所占有的资源，同时启动进程调度系统程序 schedule()重新调度，让其他就绪者占有 CPU。如果进程通过系统调用设置跟踪标志，则在系统调用返回前进入系统调用跟踪(syscall_trace())，进程状态就变为暂停状态。CPU 经重新调度给其他进程，仅当其他进程发出暂停进程信号(SIG KILL)或 SIG CONT 时，才能把暂停状态唤醒，重新插入就绪队列。

3.8.3 Linux 的进程控制

1. 进程的建立

Linux 中的绝大多数进程也是有生命期的，因创建而产生，因调度而运行，因撤销而消

失。系统启动时运行于核心模式,此时只有一个初始化进程,即0#进程。当系统初始化完毕后,初始化进程启动init进程,然后进入空闲等待的循环中。

init进程即1#进程,它完成一些系统初始化工作,如打开系统控制台、装根文件系统等。初始化工作结束后,init进程通过系统调用fork()为每个终端创建一个终端子进程为用户服务,如等待用户登录、执行Shell命令解释程序等。每个终端进程又可创建自己的子进程,从而形成一棵进程树。

在Linux系统中,系统函数fork()、vfork()和clone()都可以创建一个进程,但它们都是通过内核函数do_fork()实现的。

do_fork()函数的主要工作如下:

(1) 为新进程分配一个唯一的进程标识号PID和task_struct结构,然后把父进程中PCB的内容复制给新进程后检查用户具有执行一个新进程的必需的资源。

(2) 设置task_struct中与父进程值不同的数据成员,如初始化自旋锁、初始化堆栈信息等,同时会把新创建的子进程运行状态置为TASK_RUNNING(这里应该是就绪态)。

(3) 设置进程管理信息,根据do_fork()提供的clone_flags参数值,决定是否对父进程task_struct结构中的文件系统、已打开的文件指针等所选择的部分进行复制,增加与父进程相关联的有关文件系统的进程引用数。

(4) 初始化子进程的内核栈。通过复制父进程的上下文来初始化新进程的硬件上下文。把新进程加入到pidhash[ ]散列表中,并增加任务计数值。

(5) 启动调度程序使子进程获得运行机会。向父进程返回子进程的PID。设置子进程在系统调用do_fork()时返回0。

例3-11 调用fork()创建子进程的例子。

```
/* fork_test.c */
#include<sys/types.h>
#inlcude<unistd.h>
main()
{
    pid_t pid;                    /* 此时仅有一个进程 */
    pid=fork();                   /* 此时已经有两个进程在同时运行 */
    if(pid<0)
        printf("error in fork!");
    else if(pid==0)
        printf("I am the child process, my process ID is %d\n",getpid());
    else
        printf("I am the parent process, my process ID is %d\n",getpid());
}
```

编译并运行:

```
$gcc fork_test.c -o fork_test
$./fork_test
I am the parent process, my process ID is 1991
I am the child process, my process ID is 1992
```

在这个程序中,在语句pid=fork();之前,只有一个进程在执行这段代码,但在这条语

句之后，就变成两个进程在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是 if(pid==0)…。两个进程中，原先就存在的那个被称作父进程，新出现的那个被称作子进程。父子进程的区别除了进程标识符(PID)不同外，变量 pid 的值也不相同，pid 存放的是 fork()的返回值。fork()调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有 3 种不同的返回值：

(1) 在父进程中，fork()返回新创建子进程的 PID。

(2) 在子进程中，fork()返回 0。

(3) 如果出现错误，fork()返回一个负值。

在此程序中如果 pid 小于 0，说明出现了错误；pid==0，就说明 fork 返回了 0，也就说明当前进程是子进程，就去执行 printf("I am the child process…")，否则当前进程就是父进程，执行 printf("I am the parent process…")。

2. 进程的撤销

当进程执行完毕，即正常结束时，调用 exit()自我终止。当进程受某种信号(如 SIGKILL)的作用时，也经过执行 exit()而撤销。进程被撤销时，一方面要收回进程所占用的资源，另一方面还必须通知其父进程和子进程，对一些信号作必要的处理。

进程终止的系统调用 sys_exit 通过调用 do_exit()函数实现。do_exit()系统调用主要完成下列工作：

(1) 将进程的状态标志设为 PF_EXITING，表示进程正在退出状态。

(2) 释放分配给这个进程的大部分资源，包括内存、线性区描述符和页表、文件对象相关资源等。

(3) 向父进程发送信号，给其子进程重新找父进程。

(4) 将进程设置为 TASK_ZOMBIE(僵死)状态，使进程不会再被调度。

(5) 调用 schedule()，重新调度其他进程执行。

处于“僵死状态”的进程运行已经结束，不会再被调度，内核释放了与其相关的所有资源，但其进程控制块还没有释放，由其父进程调用 wait()函数来查询子孙进程的退出状态，释放进程控制块。

在后面的例 3-12 中，父进程调用函数 wait(0)等待子进程写入信息，子进程在写入信息后调用函数 exit(0)退出。

3. 程序的装入和执行

当父进程使用 fork()系统调用创建了子进程之后，子进程继承了父进程的正文段和数据段，从而执行和父进程相同的程序段。为了使 fork()产生的子进程可以执行一个指定的可执行文件，系统内核中开发了一个系统函数调用 exec()。它是一个调用族，每个调用函数参数稍有不同，但目的都是把文件系统中的可执行文件调入并覆盖调用进程的正文段和数据段之后执行。

3.8.4 Linux 的进程通信

Linux 提供的进程通信方式和原理与 UNIX 的通信机制一样，有管道方式、信号方式

和 System V 的 IPC 通信机制。下面分别进行介绍。

1. 管道通信方式

管道通信方式是传统的进程通信技术。管道通信技术又分为无名管道和有名管道两种类型。

无名管道为建立管道的进程及其子孙进程提供一条以比特流方式传递消息的通信管道。该管道在逻辑上被看做管道文件,在物理上则由文件系统的高速缓存区构成。发送进程利用系统调用 write(fd[1],buf,size)把 buf 中的长度为 size 字符的消息送入管道口 fd[1],接收进程则使用系统调用 read(fd[0],buf,size)从管道出口 fd[0]读出 size 字符的消息送入 buf 中。此外,管道按先进先出(FIFO)方式传递消息,且只能单向传递,如图 3-14 所示。

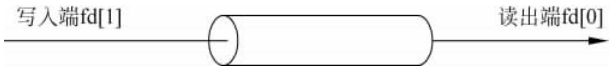

图 3-14 管道通信示意图

例 3-12 用 C 语言编写一个 C 程序,建立一个管道,同时父进程生成一个子进程,子进程向管道中写入一个字符串,父进程从管道中读出该字符串,从而实现数据的传递。程序如下:

```
#include <stdio.h>
main()
{
    int x,fd[2];
    char buf[50],s[50];
    pipe(fd);                    /* 系统调用,可建立一条同步通信管道 */
    while((x = fork()) == -1);
    if (x == 0)
    {
        sprintf(buf,"This is an example of pipe\n");
        write(fd[1],buf,50);
        exit(0);
    }
    else
    {
        wait(0);
        read(fd[0],s,50);
        printf("%s",s);
    }
}
```

在 Linux 中,管道是通过指向同一个临时 VFS 的 i 节点的两个文件 file 数据结构的文件描述符实现的。如图 3-15 所示。

VFS 中 i 节点指向内存中的一个物理页面,进程各自的 file 结构都有 f_op 操作项,分别指向管道写操作和管道读操作。当写入过程对管道进行写入操作时,数据被复制到共享

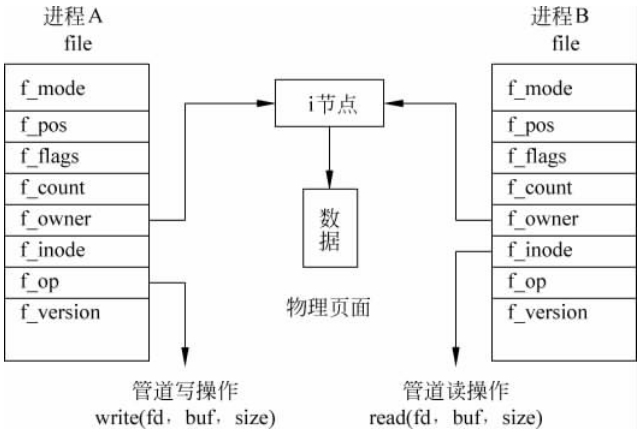

图 3-15 管道结构示意图

的数据页面中，而读取进程则从管道的共享数据页面中复制数据，从而实现了进程之间的数据传递。

Linux 必须保证对管道访问的同步，为此需要使用锁、等待队列和信号量等同步机制。当管道写入进程欲对管道写入时，该进程使用标准的文件写函数 pipe_write()。表示打开文件和打开管道的描述符用来对进程的 file 数据结构进行索引。Linux 系统调用使用由管道 file 数据结构的 f_op 属性指向的管道写过程，这个写过程用保存在表示管道的 VFS i 节点中的信息来管理写入请求。

只要管道未被读出进程加锁，系统就为写入进程对管道加锁，并将写入进程地址空间中的数据复制到共享数据页面中。若管道已被读取进程加锁，或者没有足够空间存储数据，则当前进程将进入管道 i 节点等待队列，同时进程调度程序选择其他进程投入执行。若写进程是可中断的，当有足够的空间或管道被解锁时，该进程将被读取进程唤醒。在数据写入时，管道的 VFS i 节点解锁，同时所有在该节点的等待队列上睡眠的读出进程都将被唤醒。

从管道中读出数据的过程与写入过程类似。Linux 允许进程以非阻塞的方式读出管道的内容。此时如果没有数据可读或者管道被加锁，则返回出错信息。阻塞方式则使该进程在管道 i 节点的等待队列上睡眠，直到写进程写入操作的结束。当两个进程对管道的使用结束后，管道 i 节点或共享数据页面同时被释放。

Linux 还支持有名管道(named pipe)，即 FIFO 管道。这种管道总是按先进先出的原则工作，第一个被写入的数据将首先从管道中读出。和无名管道不同的是，有名管道不是临时对象，而是文件系统中的实体，并且可以通过 mkfifo 命令来创建。进程只要拥有适当的权限就可以自由地使用有名管道。

无名管道需要先创建(建立 file 数据结构、VFS i 节点和共享数据页面)，而有名管道已经存在，使用者只需打开与关闭。在写入进程打开有名管道之前，Linux 必须让读出进程先打开此管道，任何读出进程从中读出之前必须有写入进程向其中写入数据。有名管道的使用方法与无名管道基本相同，也使用相同的数据结构和操作。

2. 信号

信号主要用来向进程发送异步的事件信号，发送信号表明要求一个进程做某件事。用

户可以用键盘中断产生信号,中断一个进程的运行,而浮点运算溢出或者内存访问错误等也可产生信号,告知相关的进程产生了异步事件。进程可以选择对某种信号所采取的特定操作,这些操作包括:

(1) 忽略信号和阻塞信号。进程可忽略产生的信号,但 SIGKILL 和 SIGSTOP 信号不能被忽略;进程可选择阻塞某些信号。

(2) 由进程处理该信号。进程本身可在系统中注册处理信号的处理程序地址,当发出该信号时,由注册的处理程序处理信号。

(3) 由内核进行默认处理。信号由内核的默认处理程序处理。大多数情况下,信号由内核进行处理。

在 Linux 内核中不存在任何机制用来区分不同信号的优先级。也就是说,多个信号发出时,进程可能会以任意顺序接收到信号并进行处理。另外,如果进程在处理某个信号之前又有相同的信号发出,则进程只能接收到一个信号。

系统在 task_struct 结构中利用两个字分别记录当前挂起的信号(signal)以及当前阻塞的信号(blocked)。挂起的信号指尚未进行处理的信号,阻塞的信号指进程当前不处理的信号。如果产生了某个当前被阻塞的信号,则该信号会一直保持挂起,直到该信号不再被阻塞为止。除了 SIGKILL 和 SIGSTOP 信号外,所有的信号均可以被阻塞,信号的阻塞可通过系统调用实现。每个进程的 task_struct 结构中还包含了一个指向 sigaction 结构数组的指针,该结构数组中的信息实际指定了进程处理所有信号的方式。如果某个 sigaction 结构中包含处理信号的例程地址,则由该处理例程处理该信号;反之,则根据结构中的一个标志或者由内核进行默认处理,也可以直接忽略该信号。通过系统调用,进程可以修改 sigaction 结构数组的信息,从而指定进程处理信号的方式。

进程不能向系统中所有的进程发送信号。一般而言,除系统和超级用户外,普通进程只会向具有相同 uid 和 gid 的进程或者处于同一进程组的进程发送信号。产生信号时,内核将进程 task_struct 的 signal 字中的相应位设置为 1,从而表明产生了该信号。系统不对其置位之前该位已经为 1 的情况进行处理,因而进程无法接收到前一次信号。如果进程当前没有阻塞该信号,并且进程正处于可中断的等待状态,则内核将该进程的状态改变为运行,并放置在运行队列中。这样,调度程序在进行调度时就有可能选择该进程运行,从而可以让进程处理该信号。

发送给某个进程的信号并不会立即得到处理,相反,只有该进程再次运行时,才有机会处理该信号。每次进程从系统调用中退出时,内核会检查它的 signal 和 block 字段,如果有任何一个未被阻塞的信号发出,内核就根据 sigaction 结构数组中的信息进行处理。处理过程如下:

(1) 检查对应的 sigaction 结构,如果该信号不是 SIGKILL 或 SIGSTOP 信号,且被忽略,则不处理该信号。

(2) 如果该信号利用默认的处理程序处理,则由内核处理该信号,否则转向第(3)步。

(3) 该信号由进程自己的处理程序处理,内核将修改当前进程的调用堆栈帧,并将进程的程序地址寄存器修改为信号处理程序的入口地址。此后,指令将跳转到信号处理程序,当从信号处理程序中返回时,实际就返回了进程的用户模式部分。

3. UNIX System V IPC 机制

Linux 支持 UNIX System V IPC(Interprocess Communication,进程间通信)机制：消息队列、信号量和共享内存。在 IPC 机制中,系统在创建这 3 种对象时就给每个对象设定了一个 ipc_perm 结构的访问权限,并返回一个标识。进程通信时必须先传递该标识,待 ipcperms()函数确认权限后才可以访问通信资源。

1) 消息队列

一个或多个进程可向消息队列写入消息,而一个或多个进程可从消息队列中读取消息,这种进程间通信机制通常使用在客户/服务器模型中,客户向服务器发送请求消息,服务器读取消息并执行相应请求。在许多微内核结构的操作系统中,内核和各组件之间的基本通信方式就是消息队列。例如,在 MINIX 操作系统中,内核、I/O 任务、服务器进程和用户进程之间就是通过消息队列实现通信的。

Linux 为系统中所有的消息队列维护一个 msgque 链表,该链表中的每个指针指向一个 msgid_ds 结构,该结构完整地描述一个消息队列。当建立一个消息队列时,系统从内存中分配一个 msgid_ds 结构并将指针添加到 msgque 链表中。

图 3-16 是 msgid_ds 结构的示意图。从图中可以看出,每个 msgid_ds 结构都包含一个 ipc_perm 结构的 msg_perms 指针,表明该消息队列的操作权限以及指向该队列所包含的消息(msg 结构)的指针。显然,队列中的消息构成了一个链表。另外,Linux 还在 msgid_ds 结构中包含了一些有关修改时间之类的信息,同时包含两个等待队列,分别用于队列的写入进程和队列的读取进程。

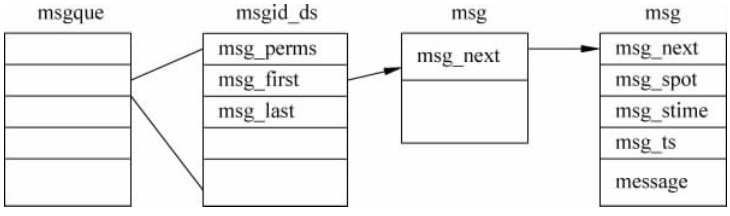

图 3-16 Linux 消息队列

消息队列是进程所读写的消息的存储空间。每当进程希望对指定队列进行写消息操作的时候,就发出系统调用：

```
sys_ipc(MSGSND,msgid,msgsz,msgflg,msgp);
```

该进程的标识 uid、gid 都首先与该队列 ipc_perm 的对应属性相比较。检查通过后,将消息复制到 msg 结构,再挂到消息队列的末尾。如果消息队列一时无法接收该消息(可能空间不够),写消息的进程暂时进入 msgid_ds 结构的写等待队列,直到这个队列的一些消息读走后,该进程才被唤醒。

读消息进程的工作进程与写消息类似。进程发出以下系统调用：

```
sys_ipc(MSGRCV,msgid,msgsz,msgflg,msgp);
```

内核系统首先检查访问权限,通过后,读取第一条消息,或读取指定类型的消息。如果

进程选择的消息不存在,则进入 msgid_ds 结构的读等待队列,当等待的消息进入队列时才被唤醒。

2) 信号量

在操作系统中,信号量最简单的形式是一个整数,多个进程可检查并设置信号量的值。这种检查和设置操作是不可被中断的,也称为原子操作。检查和设置操作的结果是信号量的当前值和设置值相加的结果,该设置值可以是正值,也可以是负值。根据检查和设置操作的结果,进行操作的进程可能会进入休眠状态,而当其他进程完成自己的检查并设置操作后,由系统检查前一个休眠进程是否可以在新信号量值的条件下完成相应的检查和设置操作。这样,通过信号量就可以协调多个进程的操作。

信号量可用来实现所谓的"关键段"。关键段指同一时刻只能有一个进程执行其中代码的代码段。也可用信号量解决经典的生产者-消费者问题。

Linux 利用 semid_ds 结构来表示 System V IPC 信号量,见图 3-17。和消息队列类似,系统中所有的信号量组成了一个 semary 链表,该链表的每个节点指向一个 semid_ds 结构。从图 3-17 可以看出,semid_ds 结构的 sem_base 指向一个信号量数组,允许操作这些信号量数组的进程可以利用系统调用执行操作。系统调用可指定多个操作,每个操作由 3 个参数指定:信号量索引、操作值和操作标志。信号量索引用来定位信号量数组中的信号量;操作值是要和信号量的当前值相加的数值。首先,Linux 按如下的规则判断是否所有的操作都可以成功:操作值和信号量的当前值相加大于 0,或操作值和当前值均为 0,则操作成功。如果系统调用中指定的所有操作中有一个操作不能成功,则 Linux 会挂起这一进程。但是,如果操作标志指定这种情况下不能挂起进程,系统调用返回并指明信号量上的操作没有成功,而进程可以继续执行。如果进程被挂起,Linux 必须保存信号量的操作状态并将当前进程放入等待队列。为此,Linux 在堆栈中建立一个 sem_queue 结构并填充该结构,将新的 sem_queue 结构添加到信号量对象的等待队列中(利用 sem_pending 和 sem_pending_last 指针),将当前进程放入 sem_queue 结构的等待队列中(sleeper),然后调用调度程序选择其他的进程运行。

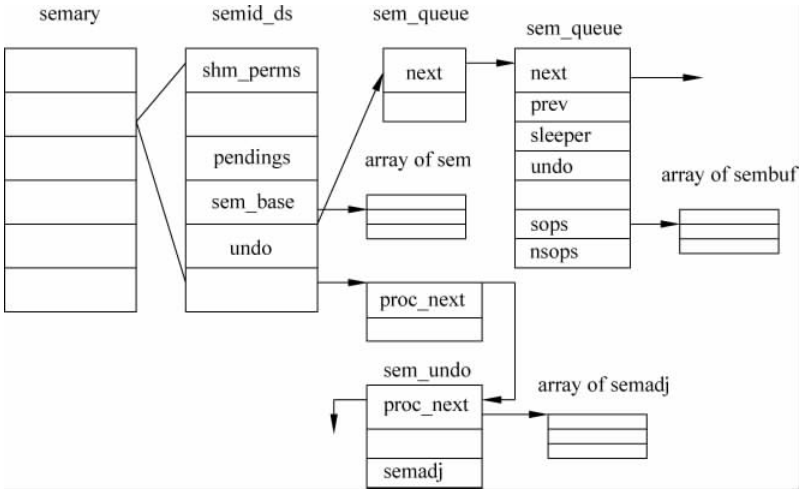

图 3-17 Linux 信号量集合

如果所有的信号量操作都成功了，当前进程可继续运行。在此之前，Linux 负责将操作实际应用于信号量队列的相应元素。这时，Linux 检查任何等待的或挂起的进程，看它们的信号量操作是否可以成功。如果这些进程的信号量操作可以成功，Linux 就会将它们从挂起队列中移去，并将它们的操作实际应用于信号量队列。同时，Linux 会唤醒休眠进程，以便在下次调度程序运行时可以运行这些进程。当新的信号量操作应用于信号量队列之后，Linux 会接着检查挂起队列，直到没有操作可成功或没有挂起进程为止。

和信号量操作相关的概念还有死锁。当某个进程修改了信号量而进入关键段之后，却因为崩溃而没有退出关键段，这时，其他被挂起在信号量上的进程永远得不到运行机会，这就是所谓的死锁。Linux 通过维护一个信号量数组的调整链表来避免这一问题。

3）共享内存

Linux 采用的是虚存、管理机制（见 5.7 节），因此，进程的虚拟地址可以映射到任意一处物理地址，这样，如果两个进程的虚拟地址映射到同一物理地址，这两个进程就可以利用这一虚拟地址进行通信。但是，一旦内存被共享之后，对共享内存的访问同步需要由其他 IPC 机制（例如信号量）来实现。Linux 中的共享内存通过访问键来访问，并进行访问权限的检查。共享内存对象的创建者负责控制访问权限以及访问键的公有或私有特性。如果具有足够的权限，也可以将共享内存锁定到物理内存中。

图 3-18 是 Linux 中共享内存对象的结构。每个新创建的共享内存区域由一个 shmid_ds 数据结构来表示。它们被保存在 shm_segs 数组中。shmid_ds 数据结构描述共享内存的大小、进程如何使用以及共享内存映射到其各自地址空间的方式。由共享内存创建者控制对此内存的存取权限以及其键是公有还是私有。如果它有足够权限，还可以将此共享内存加载到物理内存中。每个使用此共享内存的进程必须通过系统调用将其连接到虚拟内存上。这时进程创建新的 vm_area_struct 来描述此共享内存。进程可以决定此共享内存在其虚拟地址空间的位置，或者让 Linux 选择一块足够大的区域。新的 vm_area_struct 结构将被放到由 shmid_ds 指向的 vm_area_struct 链表中。通过 vm_next_shared 和 vm_prev_shared 指针将它们连接起来。虚拟内存在连接时并没有创建，而在进程访问它时才创建。和消息队列及信号量类似，Linux 中也有一个链表维护着所有的共享内存对象。

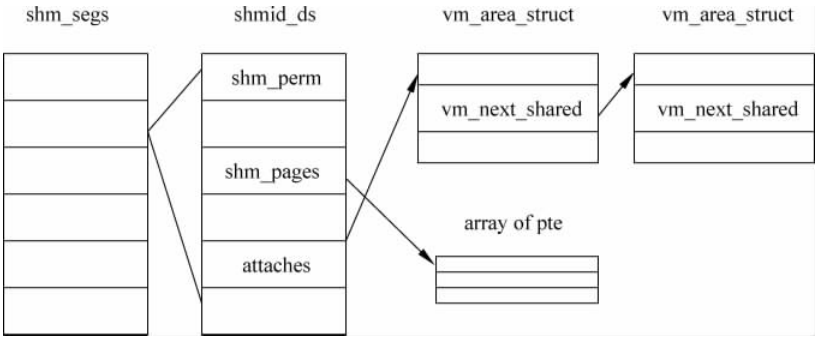

图 3-18 Linux 共享内存

某个进程第一次访问共享虚拟内存时将产生页故障。这时，Linux 找出描述该内存的 vm_area_struct 结构，该结构中包含用来处理这种共享虚拟内存的处理函数地址。共享内

存页故障处理代码在 shmid_ds 的页表项链表中查找,以便查看是否存在该共享虚拟内存的页表项。如果没有,系统将分配一个物理页并建立页表项。该页表项加入 shmid_ds 结构的同时也添加到进程的页表中。此后,当另一个进程访问该共享内存时,共享内存页故障处理程序将使用同一物理页,而只是将页表项添加到这一进程的页表中。这样,前后两个进程就可以通过同一物理页进行通信。

当进程不再共享此虚拟内存时,进程和共享内存的连接将被断开。如果其他进程还在使用这个内存,则此操作只影响当前进程。其对应的 vm_area_struct 结构将从 shmid_ds 结构中删除并回收。当前进程对应此共享内存地址的页表入口也将被更新并置为无效。当最后一个进程断开与共享内存的连接时,当前位于物理内存中的共享内存页面将被释放,同时释放的还有此共享内存的 shmid_ds 结构。

习题

1. 程序的顺序执行和并发执行各有什么特点?
2. 进程的定义是什么? 为什么要引入进程?
3. 进程和程序有什么区别?
4. 进程有哪些特征? 基本特征是什么?
5. 进程的静态描述由哪几部分组成?
6. 为什么说进程控制块是操作系统感知进程存在的唯一标志?
7. 进程在运行过程中有哪些基本状态? 各状态之间转换的条件是什么?
8. 试述引起进程创建的主要事件。
9. 试述引起进程被撤销的主要事件。
10. 试述引起进程阻塞或唤醒的主要事件。
11. 试述创建进程原语的主要工作。
12. 试述撤销进程原语的主要工作。
13. 试述进程阻塞原语的主要工作。
14. 试述进程唤醒原语的主要工作。
15. 什么是临界资源? 什么叫临界区? 试举例说明。
16. 并发进程间的制约有哪两种? 引起制约的原因是什么?
17. 什么是进程间的互斥关系? 什么是进程间的同步关系?
18. 什么是原语? 用户进程通过什么方式访问内核原语?
19. 为什么说在阻塞原语的最后必须转入进程调度?
20. 互斥机构应遵循的准则是什么?
21. 简述 P、V 原语的主要操作。
22. P、V 操作的物理意义是什么?
23. 如何用信号量实现进程间的互斥? 举例说明。
24. 什么叫进程通信? 进程通信有哪两类?
25. 根据数据存取的方式,进程高级通信方式有哪些?
26. 为什么在操作系统中引入线程?

27. 试述线程与进程的区别与联系。

28. 试述用户态线程和核心态线程的优缺点。

29. 试画出下面 6 条语句的前趋图，并用 P、V 操作描述它们之间的同步关系。

S1：X1＝a＊a；

S2：X2＝3＊b；

S3：X3＝5＊a；

S4：X4＝X1＋X2；

S5：X5＝b＋X3；

S6：X6＝X4/X5；

30. 有 3 个进程 PA、PB、PC 合作解决文件打印问题：PA 将文件记录从磁盘读入主存的缓冲区 1，每执行一次读一个记录；PB 将缓冲区 1 的内容复制到缓冲区 2，每执行一次复制一个记录；PC 将缓冲区 2 的内容打印出来，每执行一次打印一个记录。缓冲区的大小等于一个记录大小。请用 P、V 操作来保证文件的正确打印。

31. 桌上有一个空盘，允许存放一只水果。爸爸可向盘中放苹果，也可向盘中放橘子。儿子专等吃盘中的橘子，女儿专等吃盘中的苹果。规定当盘中空时一次只能放一只水果供吃者取用。请用 P、V 原语实现爸爸、女儿、儿子三个并发进程的同步关系。

32. 有一个阅览室，共有 100 个座位。读者进入时必须先在一张表上登记，该登记表每一座位列一表目，包括座号和读者姓名。读者离开时要消掉登记内容。试用 P、V 原语描述读者进程间的同步关系。

33. Linux 进程有哪几种基本状态？各个状态是如何转换的？

34. Linux 中如何创建进程？创建进程时需要做哪些工作？

35. 编写一个程序，使用系统调用 fork()生成 3 个子进程，并使用系统调用 pipe()创建一个管道，使得这 3 个子进程和父进程共用一个管道进行通信。