第4章 进程管理

4.1 学习引导

【单元概述】

在现代计算机系统中,为了提高系统资源的利用率,大都采用多道程序设计技术(即允许多个程序同时进入内存并发运行)。这些程序共享系统资源,同时对资源也存在竞争关系。操作系统必须对各种资源进行合理的分配和调度,处理并发程序之间的制约关系及通信关系。而操作系统的进程很好地解决了上述问题。

本章以进程为核心,描述进程与程序的区别、进程的基本特性,进程从创建到结束的整个状态及其转换,以及解决进程之间的通信、在竞争资源中引起的同步、互斥、死锁等问题的方法。

本章将各个知识点贯穿在四个不同的项目之中,其中进程管理是基础性项目,是其他三个项目的基础。进程管理及生产者消费者问题两个项目都是采用C语言进行模拟的项目;而读者写者问题及哲学家进餐问题两个项目是使用Linux API实现的项目,目的在于使读者了解真实的进程创建及通信机制。同时本章涉及到大量的Linux源代码程序,读者如能结合理论知识,认真阅读Linux内核源代码将受益匪浅。本章的知识架构如图4.1所示。

【知识要点及掌握程度】

了解程序的顺序执行及程序与进程的区别;
理解进程的基本状态及其转换,进程的描述方法(PCB);
理解进程控制的基本方法及进程间通信的方法;
运用进程的同步与互斥解决资源竞争及死锁问题。

【能力要点及重要程度】

(1)计算机专业基础知识:理解进程管理方法、临界区、进程同步与互斥、线程等相关概念。(重要)

(2)分析问题:针对相关具体问题,要能够运用进程同步与互斥的方法,逐步给出解决问题的方案。(重要)

东软电子出版社

(3)发现问题和表述问题:在具体的任务、方案或算法中,能够发现问题并给出评价。(中等)

(4)软件实现过程:能完成实验中程序编写,通过编写代码实现四级项目功能及完成作业内的项目。(中等)

(5)查询印刷资料和电子文献:了解常用的进程同步与互斥的解决方法。(中等)

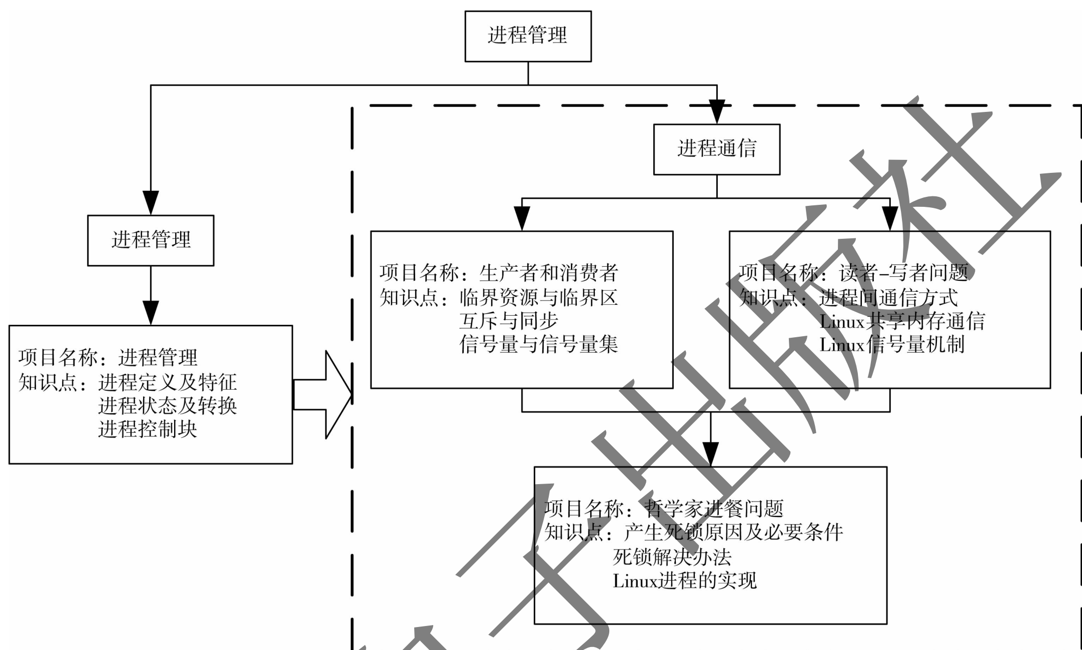

图 4.1　本章知识框架

【教学重点与难点】

重点:进程的概念、状态及其转换;进程通信、互斥与同步、进程死锁。

难点:运用进程互斥与同步解决实际问题。

【教学设计与实施方法】

本章主要采用讲授教学法、练习教学法、演示教学法、实验教学法、自主学习教学法。讲授教学法通过教师课堂讲授实施,练习教学法通过学生课后作业实施,演示教学法通过教师演示实验项目实施,实验教学法通过学生自主完成实验实施,自主学习教学法通过学生课后完成。

【实践环节设计】

四级项目:进程管理的模拟实现。

四级项目:生产者—消费者问题。

四级项目:利用共享内存实现读者—写者问题。

四级项目:哲学家就餐问题。

四级项目:司机与售票员问题。

东软电子出版社

4.2　进程管理

多道程序在执行时,需要共享系统资源,从而导致各程序在执行过程中出现相互制约的关系,程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中发生的,是动态的过程,而传统的程序本身是一组指令的集合,是一个静态的概念,无法描述程序在内存中的执行情况,即我们无法从程序的字面上看出它何时执行,何时停顿,也无法看出它与其他执行程序的关系。因此,程序这个静态概念已不能如实反映程序并发执行过程的特征。为了深刻描述程序动态执行过程的性质,人们引入了“进程(Process)”的概念。

完成本节的学习后,应能独立完成单元项目进程管理的模拟实现。

4.2.1　程序、程序的顺序执行及其特征

1.程序(Program)

程序是为实现特定目标或解决特定问题而用计算机语言编写的命令序列的集合,是用汇编语言、高级语言等开发编制出来的可以运行的文件,在计算机中称可执行文件;程序是由序列组成的,告诉计算机如何完成一个具体的任务。可见,程序是一个静态的概念,程序只能在处理器上运行才能完成具体的任务。程序在计算机上运行具有两种典型的方式:顺序执行和并发执行。

2.前趋图的定义

为了描述一个程序的各部分(程序段或语句)间的依赖关系,或者是一个大的计算的各个子任务间的因果关系,我们常常采用前趋图方式。前趋图中的每个结点可以表示一条语句、一个程序段或一个进程,结点间的有向边表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“→”。

如果(Pi,Pj)∈→,表示在 pj 开始前 pi 必须完成。可写成 Pi→Pj,Pi 是 Pj 的直接前趋,Pj 是 Pi 的直接后继。

例如,图 4.2 表示了具有九个结点的前趋图:

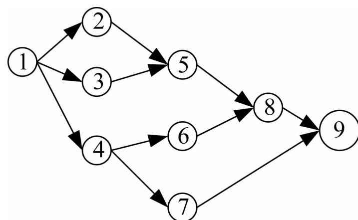

图 4.2　具有 9 个节点的前趋图表示

在图 4.2 中 p1 为初始结点,p9 为终止节点。每个节点除表示先后关系外,还可以表示一定的权重。在该前趋图,存在下面的前趋关系:

P1→P2,P1→P3,P1→P4,P2→P5,P3→P5,P4→P6,P4→P7,P5→P8,P6→P8,P7→P9,P8→P9;

或表示为:

东软电子出版社

p={P1,P2,P3,P4,P5,P6,P7,P8,P9}

={(P1,P2),(P1,P3),(P1,P4),(P2,P5),(P3,P5),(P4,P6),(P4,P7),(P5,P8),(P6,P8),(P7,P9),(P8,P9)}

注意：

前趋图中必须不能存在循环关系。

3.程序的顺序执行

通常一个程序可分成若干个程序段，它们必须按照某种先后次序执行，仅当前一操作执行后，才能执行后继操作。程序的这种执行方式称为程序的顺序执行。

例如，从终端输入两个数字，计算这两个数字的和，并通过打印机打印出结果。该程序可以用 3 个程序段来表示：数字输入(I)、计算(C)、打印(P)。在顺序执行的情况下这些操作的先后次序可用图 4.3 来表示：

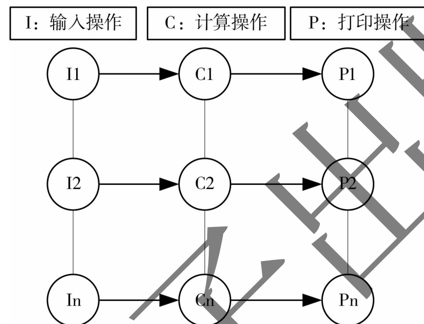

图 4.3　程序顺序执行的表示

如图 4.3 所示，在进行计算时，总是先输入用户的程序和数据，然后进行技术，最后将结果打印出来。并且在第一组输入、计算、打印序列执行结束后，才能继续第二组输入、计算、打印操作序列，并依次类推，直至最后执行完第 n 组序列。

又如存在如下的程序片段：

```
S1：a = 10；
S2：b = a+10；
S3：c = b。
```

则语句 S2 必须在 a 被赋值后才能执行，S3 也只能在 b 被赋值后才能执行。

4.程序顺序执行的特征

(1)顺序性：程序顺序执行时，其执行过程可看做一系列严格按程序规定的状态转移的过程，也就是每执行一条指令，系统就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：顺序执行的最终结果可再现是说它与执行速度无关。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。

4.2.2　程序的并发执行及其特征

1.程序并发执行

多道程序(即多个程序同时存在于内存中)在同一时间间隔内同时执行，即程序段的执行在

东软电子出版社

时间上是重叠的。

例如在上例中的输入、计算、打印三个程序对一批作业进行处理时,存在以下的前趋关系:Ii→Ci,Ii→Ii+1,Ci→Pi,Ci→Ci+1,Pi→Pi+1。可用图4.4表示如下:

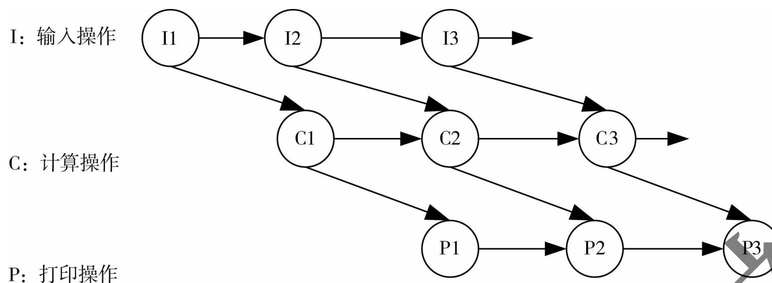

图4.4 程序的并发执行

从图4.4可以看出,Ii、Ci、Pi在时间上是存在着先后关系的,而Ii+2、Ci+1、Pi只是部分有序的,这种部分有序使操作的并发执行成为了可能。例如I3、C2、P1等操作的执行在时间上互相重叠,为并发执行提供了可能。

2.程序并发执行的特征

(1)间断性:程序并发执行时,由于它们共享资源或程序之间相互合作完成一项共同任务,因而使程序之间相互制约。

(2)失去封闭性:多个程序共享系统中的资源,而这些资源的状态将由多个程序来改变。

(3)不可再现性:由于失去封闭性,也将导致失去可再现性;外界环境在程序的两次执行期间发生变化,失去原有的可重复特征。

4.2.3 进程的定义及其特征

由程序的并发执行可以知道,多道程序在执行时需要共享系统资源,从而使各程序在执行过程中出现相互制约的关系,程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中发生的,是动态的过程,而传统的程序本身是一组指令的集合,是一个静态的概念,无法描述程序在内存中的执行情况,即我们无法从程序的字面上看出它何时执行,何时停顿,也无法看出它与其他执行程序的关系,因此,程序这个静态概念已不能如实反映程序并发执行过程的特征。为了深刻描述程序动态执行过程的性质,人们引入“进程(Process)”概念。

进程的概念是20世纪60年代初首先由麻省理工学院的MULTICS系统和IBM公司的CTSS/360系统引入的。它是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源,是一个动态的概念,是一个活动的实体。它不只是程序的代码,还包括当前的活动,通过程序计数器的值和处理寄存器的内容来表示。进程的通用定义如下:

1.进程的定义

进程是一个具有一定独立功能的程序关于某个数据集合上的一次运行活动。它是操作系统动态执行的基本单元,在传统的操作系统中,进程既是基本的分配单元,也是基本的执行单元。

进程的概念主要有两点:第一,进程是一个实体。每一个进程都有它自己的地址空间,一般情况下,包括文本区域(text region)、数据区域(data region)和堆栈(stack region)。文本区域存储处理器执行的代码;数据区域存储变量和进程执行期间使用的动态分配的内存;堆栈区域存

东软电子出版社

储着活动过程调用的指令和本地变量。第二,进程是一个“执行中的程序”。程序是一个没有生命的实体,只有处理器赋予程序生命时,它才能成为一个活动的实体,我们称其为进程。

进程是操作系统中最基本、最重要的概念,是多道程序系统出现后,为了刻画系统内部出现的动态情况,描述系统内部各道程序的活动规律引进的一个概念,所有多道程序设计操作系统都建立在进程的基础上。

2.进程的特征

(1)动态性。进程是进程实体的执行过程,因此,动态性是进程最基本的特征。进程由创建而产生,由调度而执行,因得不到资源而暂停执行,并因撤销而消亡,可见,进程具有一定的生命周期。

(2)并发性。这是指多个进程实体,共存于内存中,能在一段时间段内同时执行。并发性是进程的重要特征,同时也是操作系统的重要特征。提高并发性,可以提高系统的效率。

(3)独立性。进程是一个能独立运行的基本单位,同时也是系统中独立获得资源和独立调度的基本单位。凡未建立进程的程序,都不能作为一个独立的单位参加运行。

(4)异步性。这是指进程按各自独立的、不可预知的速度向前推进,或者说,进程按异步方式运行。这一特征将导致程序执行的不可再现性,因此,在 OS 中必须采取某种措施来保证各程序之间能协调运行。

(5)结构特征。从结构上看,进程实体是由程序段、数据段及进程控制块 3 部分组成,也称这 3 部分为进程映像。

3.进程与程序区别

(1)程序是进程的静态文本,进程是执行程序的动态过程。

(2)一个进程可以执行一个或多个程序,几个进程可以同时执行一个程序。

(3)程序可作为软件资源长期保存,进程只是一次执行过程,是暂时的。

操作系统引入进程的概念的原因主要有以下两点:

(1)从理论角度看,是对正在运行的程序过程的抽象。

(2)从实现角度看,是一种数据结构,目的在于清晰地刻划动态系统的内在规律,有效管理和调度进入计算机系统主存储器运行的程序。

4.进程小结

到此,我们将进程的相关知识总结如下:

(1)进程是一个可执行的程序,包括初始代码和数据。

(2)进程占有独立的用户地址空间及系统资源,在进程创建时由操作系统分配。

(3)进程至少有一个执行栈区,用来保存执行现场的信息。

(4)进程是动态的、有生命周期的活动;是可以和别的进程并发执行的活动;是系统进行资源分配和调度的一个独立单位。

4.2.4　进程的基本状态及其转换

进程在它存在的过程中,由于系统中各进程并行运行及相互制约,使得他们的状态不断发生变化。大多数教材将进程状态归结为运行、就绪、阻塞三种基本状态,本教材结合 Linux 操作系统将进程状态归结为创建、运行、就绪、阻塞、挂起以及结束等六种状态。

东软电子出版社

1.创建状态(New)

进程正在被创建,但还没有获得相关的资源。

2.就绪状态(Ready)

进程已经获得了除 CPU 以外的所有资源,一旦获得 CPU 即可运行。系统中可能存在多个进程处于就绪状态,通常将它们组成一个队列,称为就绪队列。

3.运行状态(Running)

进程获得了 CPU 正在运行。在但处理机系统中,某一时刻仅有一个进程占有 CPU 并处于运行状态;而在多处理机系统中,某一时刻则可能有多个进程分别占有不同的 CPU 而同时处于运行状态。

4.阻塞状态又称等待状态(Blocked)

正在执行的进程由于不能获得需要的资源或其他原因而无法继续执行,进程只能放弃 CPU 而进入暂停状态,即进程执行受到了阻塞。

5.结束状态(Exit)

由于进程结束或其他原因,进程正在从系统中消失的状态。

6.挂起(Suspend)

进程除了具有上述五种状态外,还具有挂起状态,即按照一定的算法将内存中处于阻塞状态的进程暂时交换到外存(对应于挂起操作:Suspend),并插入到相应的挂起队列中,从而空出内存空间以调用位于外存挂起队列中将要运行的进程(对应于激活操作:Active),从而实现虚拟存储管理功能,挂起又分为就绪挂起和阻塞挂起。图 4.5 表示出了进程的各种状态及其相互间的转换关系。

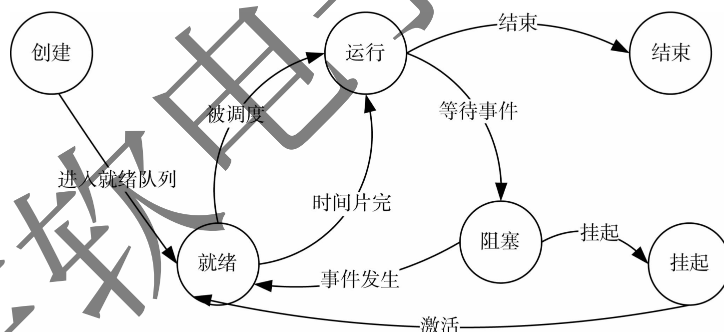

图 4.5 进程状态转换图

4.2.5 进程切换

进行进程切换就是从正在运行的进程中收回处理器,然后再使待运行进程来占用处理器。

这里所说的从某个进程收回处理器,实质上就是把进程存放在处理器的寄存器中的中间数据找个地方存起来,从而把处理器的寄存器腾出来让其他进程使用。那么被中止运行进程的中间数据应该存在何处?当然这个地方应该是进程的私有堆栈。

让进程来占用处理器,实质上是把某个进程存放在私有堆栈中寄存器的数据(前一次本进程被中止时的中间数据)再恢复到处理器的寄存器中去,并把待运行进程的断点送入处理器的程序指针 PC,于是待运行进程就开始被处理器运行了,也就是这个进程已经占有处理器的使用权了。

东软电子出版社

这就像多个同学要分时使用同一张课桌一样，所谓要收回正在使用课桌同学的课桌使用权，实质上就是让他把属于他的东西拿走；而赋予某个同学课桌使用权，只不过就是让他把他的东西放到课桌上罢了。

在切换时，一个进程存储在处理器各寄存器中的中间数据叫做进程的上下文，所以进程的切换实质上就是被中止运行进程与待运行进程上下文的切换。在进程未占用处理器时，进程的上下文是存储在进程的私有堆栈中的。

4.2.6　进程控制块 PCB

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 PCB，它是进程存在的唯一标志。

进程控制块是操作系统中最重要的数据结构，每个进程控制块都包含操作系统所需的所有进程信息。在创建一个进程时，应首先创建进程的 PCB，然后根据 PCB 中的信息对进程实施有效的管理和控制。PCB 是进程存在的唯一标志，一般常驻内存或部分常驻内存。为了保证 PCB 的安全，一般存储在系统的数据区中。当进程运行结束，系统就回收其 PCB，进程也就随之消亡。

一般来说，PCB 记录了进程的全部控制信息，内容较多。大体上可以按照功能分成 4 个组成部分：进程标识符信息、处理机状态信息、进程调度信息和进程控制信息。

1.进程标识信息

进程标识信息用于唯一地标识一个进程。一个进程通常有以下两种标识符：

(1)进程内部标识符，是操作系统为每个进程赋予的一个唯一的数字标识符，它通常为一个进程的序号，设置内部标识符的目的是为了方便系统使用。

(2)进程外部标识符，由创建进程者自己定义，通常有字母、数字组成，目的是为用户访问进程提供方便。

为了描述进程间的家族关系，通常还设有父进程标识和子进程标识。此外，还设有用户名或用户标识号表示该进程属于哪个用户。

2.处理机状态信息

进程运行时的许多信息均存放在处理机的各个寄存器中。当进程运行被中断时应保护 CPU 现场，即将有关寄存器中的内容保存到 PCB 中。以便在中断返回时能够恢复现场。这些信息包括：通用寄存器、指令计数器、程序状态字寄存器、用户堆栈指针等。

3.进程调度信息

主要用于操作系统调度进程占用处理机，主要包括以下四项信息：

(1)进程状态：标识进程当前的状态，例如进程处于运行、就绪、阻塞等。

(2)进程优先级：用于表示进程获得处理机的优先级别，优先级高的进程应先获得处理机。

(3)进程调度所需要的其他信息，这些信息为进程调度算法提供依据，例如进程等待时间、进程运行时间等。

东软电子出版社

(4)事件:进程由一种状态变化为另一种状态发生的原因。

4.进程控制信息

进程控制信息包括:

(1)程序和数据的地址,即组成进程的程序和数据所在内存或外存中的地址,以便在调度该进程执行时能找程序和数据。

(2)进程同步和通信机制,指实现进程同步和通信时所需采用的方式,例如消息队列、信号量等,他们可以全部或部分存放在 PCB 中。

(3)资源清单,列出进程除 CPU 外所需的全部资源以及已分配给该进程的资源。

(4)有关数据结构链接信息。进程可以连接到一个进程队列中或链接到相关的其他进程中。例如,同一优先级的等待进程被链接成一个队列,一个进程可以链接到它的父子进程。

4.2.7 进程控制块的组织方式

系统中有许多进程,为了对所有进程进行有效的管理,将处于同一状态的进程组织在一起。常见的组织方式有:线性方式、链接方式和索引方式。

1.线性方式

根据操作系统中允许同时存在的最大进程数,开辟静态存储空间,把所有进程的 PCB 都放在这个表中。这种方式存在的缺点是明显的:浪费空间也降低了调度的效率。

2.链接方式

用 PCB 中的链接字把具有相同进程状态的 PCB 链接成一个队列,并在操作系统中用相应队列的链头指针变量指出队列中的第一个 PCB 的头地址。这样可以形成就绪 PCB 队列、阻塞 PCB 队列和空闲 PCB 队列等。

图 4.6 描述了一种链接队列的组织方式,图中链接字值为 0,表示该 PCB 是相应 PCB 队列中的链尾 PCB。

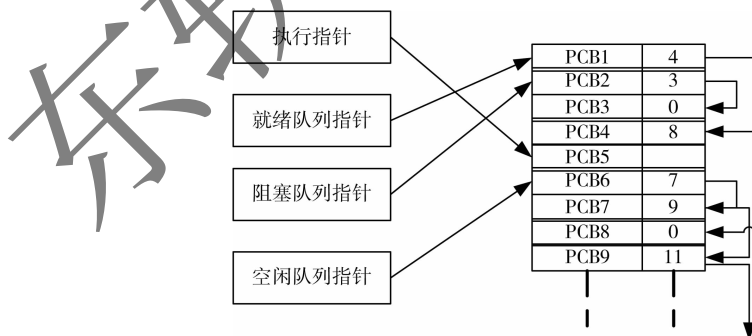

图 4.6 链接方式 PCB 组织

3.索引方式

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。如图 4.7 所示,在索引方式中,PCB 中不必设置 PCB 所属状态

东软电子出版社

的 PCB 队列链接字。

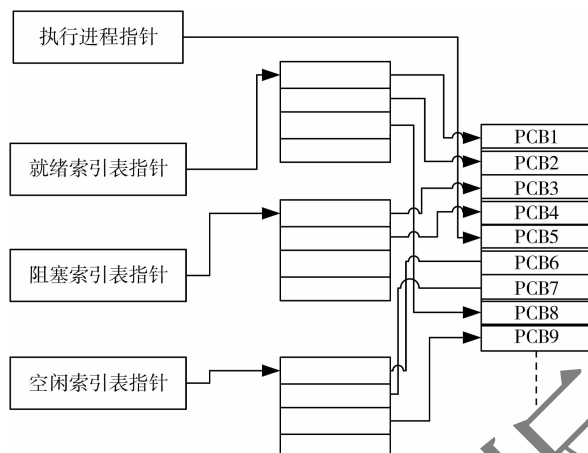

图 4.7　索引方式 PCB 组织

4.2.8　进程控制

进程控制的主要任务是对系统中所有进程从产生、存在到消亡的全过程实行有效的管理和控制。进程控制一般是由操作系统的内核来实现,内核在执行操作时往往是通过执行各种原语操作来实现的。

1.进程图

进程图描述系统中各进程之间的关系,可以使用一棵有向树来表示,树中的每个结点代表一个进程,一棵树表示一个家族,根结点为该家族的祖先(Ancestor),类似于 Linux 系统中的用户进程 init。树中父子节点关系表示了进程的父子关系。

注意:

进程图和前趋图之间的差异:

(1)前趋图描述的是任务(或进程)之间的前趋关系;只有在前趋进程完成后,其后继进程才能运行。

(2)在进程图中,创建者和被创建者可以并发执行,也可以是父进程等待其所有的子进程结束后再执行,这完全取决于创建原语和创建者的需要。

常见的进程原语包括:进程创建、进程撤销、进程阻塞、进程唤醒、进程挂起与激活等。

2.进程的创建

进程创建方式有以下两种:

(1)由系统程序模块统一创建,例如在批处理系统中,由操作系统的作业调度程序为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,父进程创建子进程以完成并行工作。

由系统统一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。而在父进程创建的进程之间则存在隶属关系,且互相构成树型结构的。属于某个家族的一个进程可以继承其父进程所拥有的资源。

东软电子出版社

无论是哪种方式创建进程,都必须调用创建原语来实现。创建原语扫描系统的 PCB 链表,在找到一定 PCB 链表之后,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:进程名、进程优先级、进程正文段起始地址、资源清单等。创建原语的流程可以用图 4.8 来表示。

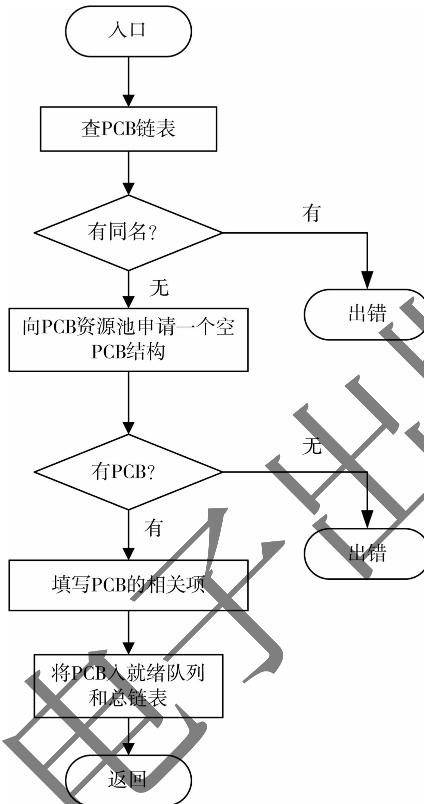

图 4.8 进程创建原语流程

3.进程的撤消

进程撤消或称为"终止"。在以下几种情况下会导致进程被撤消:

(1)该进程已完成所要求的功能而正常终止。

(2)由于某种错误导致非正常终止。

(3)祖先进程要求撤消某个子进程。

无论哪一种情况导致进程被撤消,进程都必须释放它所占用的各种资源和 PCB 结构本身,以利于资源的有效利用。当然,一个进程所占有的某些资源在使用结束时可能早已释放。

撤消原语首先检查 PCB 进程链或进程家族,寻找所要撤消的进程是否存在。如果找到了所要撤消的进程的 PCB 结构,则撤消原语释放该进程所占有的资源之后,把对应的 PCB 结构从进程链或进程家族中摘下并返回给 PCB 空队列。如果被撤消的进程有自己的子进程,则撤消原语先撤消其子进程的 PCB 结构并释放子进程所占用的资源之后,再撤消当前进程的 PCB 结构和释放其资源。如图 4.9 所示。

东软电子出版社

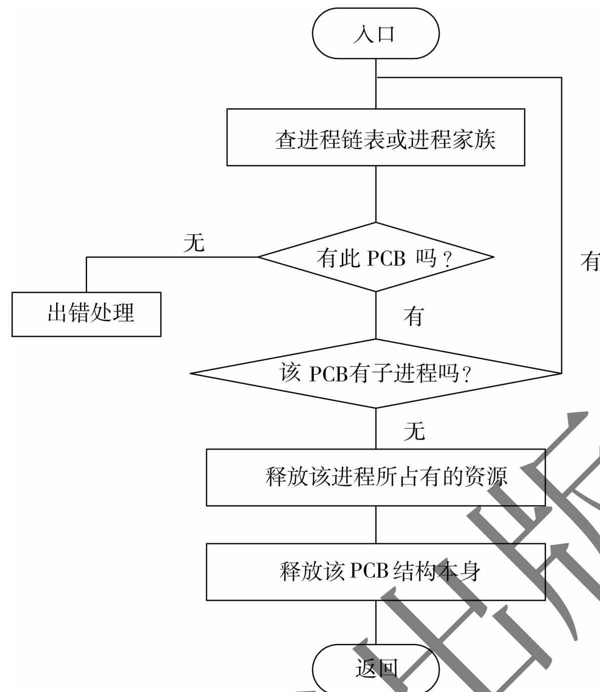

图 4.9　进程撤销原语流程

4. 进程的阻塞

阻塞原语在一个进程期待某一事件(例如键盘输入数据、写盘、其他进程发来的数据等)发生,但发生条件尚不具备时,被该进程自己调用来阻塞自己。阻塞原语在阻塞一个进程时,由于该进程正处于执行状态,故应先中断处理机和保存该进程的 CPU 现场。然后将被阻塞进程置“阻塞”状态后插入等待队列中,再转进程调度程序选择新的就绪进程投入运行。该过程可以用图 4.10 来表示。

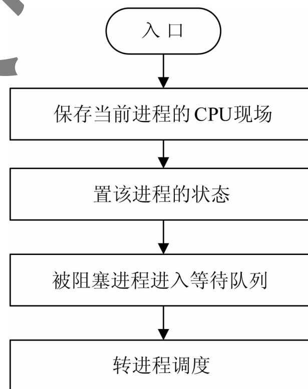

图 4.10　进程阻塞原语流程

5. 进程的唤醒

当等待队列中的进程所等待的事件发生时,等待该事件的进程都将被唤醒。唤醒一个进程有两种方法: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,系统进程统一控制事件的发生并将“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唤醒。由事件发生进程唤

东软电子出版社

醒时,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,将被唤醒进程置为就绪状态之后,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,以便让调度程序有机会选择一个合适的进程执行。该过程可以用图 4.11 来表示。

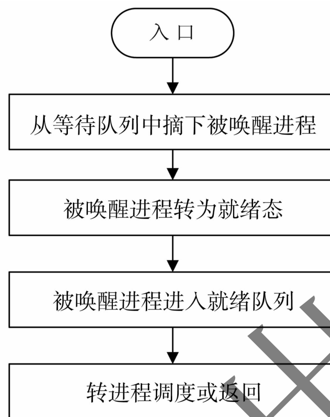

图 4.11 进程唤醒原语流程图

4.3 进程互斥与同步及信号量机制

多进程的系统中避免不了进程间的相互作用关系。本讲将介绍进程间的两种主要关系——同步与互斥,然后着重讲解解决进程同步的常用机制之一——信号量机制。

进程互斥是进程之间发生的一种间接性作用,一般是程序不希望的。通常的情况是两个或两个以上的进程需要同时访问某个共享变量。我们一般将发生能够访问共享变量的程序段成为临界区。两个进程不能同时进入临界区,否则就会导致数据的不一致,产生与时间有关的错误。解决互斥问题应该满足互斥和公平两个原则,即任意时刻只能允许一个进程处于同一共享变量的临界区,而且不能让任一进程无限期地等待。互斥问题可以用硬件方法解决,我们不作展开;也可以用软件方法,这将会在本讲作详细介绍。

完成本节的学习后,应能独立完成单元项目生产者消费者问题。

4.3.1 临界资源和临界区

1.临界资源(Critical Resource)

众所周知,计算机系统中的资源是有限的,为了提高系统资源的利用率,同时满足多个并发进程运行的需要,操作系统引入了资源共享机制,即允许多个进程同时访问系统中的共享资源。但是,系统中许多资源是不允许同时访问的,如物理设备:输入机、打印机等,软件资源:变量、表格、队列等。并发进程对这些资源的使用是轮流进行的。也就是当某个进程使用这些资源时,其他进程则不能使用该资源,即各并发进程对这些资源的使用是互斥的。这种一次仅允许一个

东软电子出版社

进程访问的共享资源称为临界资源(简称 CR)。

2.临界区(Critical Section)

当若干个进程均要使用同一临界资源时,它们必须以互斥的方式进入。人们把程序中使用某一临界资源的那段代码称为程序的临界区。可见,临界区是相对于某一临界资源的,而且在某一时刻仅允许一个程序处于临界区。

4.3.2　进程互斥与同步的概念

1.进程互斥与同步的概念

进程共享资源而导致了进程的相互制约,这种相互制约关系可以分为两种:

(1)互斥关系(也称间接制约关系)。

进程互斥是进程之间发生的一种间接性作用,一般是程序不希望的。通常的情况是两个或两个以上的进程需要同时访问某个共享变量(临界区)。两个进程不能同时进入临界区,否则就会导致数据的不一致,产生与时间有关的错误,即进程互斥是进程间因相互竞争使用临界资源而产生的制约关系。解决互斥问题应该满足互斥和公平两个原则,即任意时刻只能允许一个进程处于同一共享变量的临界区,而且不能让任一进程无限期地等待。互斥问题可以用硬件方法解决,我们不作展开,本教材只介绍采用软件方法解决进程互斥。

(2)同步关系(也称直接制约关系)。

进程同步是进程之间直接的相互作用,是完成同一任务的进程之间,因为需要在某些位置上协调它们的工作而相互等待、相互交换信息所产生的制约关系。实现进程同步的机制称为同步机制,如信号量操作、加锁操作等。如果系统中存在两个进程:计算进程和打印进程,它们公用一个打印缓冲区。计算进程将其计算结果送入打印缓冲区,打印进程从中取出并送到打印机上打印出来。显然计算进程和打印进程为一对协作进程,它们公共协作完成待求解问题的计算和打印工作。

进程同步与互斥的典型例子是公共汽车上司机与售票员的合作。售票员具有如下的动作序列:关车门、售票、开车门,司机的动作序列如下:启动车辆、正常行驶、到站停车。司机和售票员的动作需要一定的协调。其中主要的两点是:

(1)司机开车的时候,售票员不能开门(这里体现的是进程的互斥问题),车停之后,由司机通知售票员开门(这里体现的是进程的同步问题)。

(2)车门开着的时候,司机不能开车,等售票员把车门关上之后,由售票员通知司机开车。

2.实现进程互斥和同步应遵循的原则

一组并发进程互斥、同步执行时必须满足如下原则:

(1)空闲让进。当没有进程处于临界区时,任何要求进入临界区的进程应允许立即进入。

(2)忙则等待。并发进程中的若干个进程申请进入临界区时,只允许一个进程进入。当已有进程进入临界区时,其他申请进入临界区的进程必须等待,以保证对临界资源的互斥访问。

(3)有限等待。当多个进程访问临界资源时,应保证在有限的时间内进入自己的临界区,而不应相互阻塞,以至于各个进程都不能进入临界区。

(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,以免进入陷入“忙等”状态。

东软电子出版社

4.3.3 信号量机制

进程的互斥与同步是操作系统的重要任务之一,1965 年由荷兰的 Dijkstra 提出的信号量机制,是一种有效的进程同步工具,已被广泛应用于单处理机系统、多处理机系统以及计算机网络。在应用中,信号量机制又得到了很大的发展,从经典信号量,到计数型信号量,发展为“信号量集”机制。信号量(Semaphore)也叫做信号灯,是一种特定类型的数据结构。在该机制中申请和释放临界资源的两个原语操作为 wait 操作和 signal 操作,有时也称为 P 操作和 V 操作。wait(S)和 signal(S)有时也称为 P 原语操作和 V 原语操作,P 和 V 相对于英文中的 Pass 和 Increment 的意思。原语操作是不可中断的程序段,如果将信号量看作共享变量,则 pv 操作为其临界区,多个进程不能同时执行,一般用硬件方法保证。一个信号量只能置一次初值,以后只能对该信号量进行 p 操作或 v 操作。由此也可以看到,信号量机制必须有公共内存,不能用于分布式操作系统,这是它最大的弱点。

1.经典信号量机制

也称为整型信号量,信号量的值表示当前系统中可用的该类临界资源的数量,如使用 S 表示信号量,则 S 的值的意义为:

S>0,表示系统中空闲的该类临界资源的个数;

S=0,表示系统中该类临界资源刚好全部被占用,而且没有进程在等待该临界资源;

S<0,S 的绝对值表示系统中等待该类临界资源的进程的个数。

经典信号量的 P(S)、V(S)操作可以描述为:

```
P(S):
wile S≤ 0 该进程等待
   S = S - 1;
V(S):
   S = S+1;
```

P(S)操作的特点是,如果在 S ≤ 0 条件系执行 P 操作,必然会陷入“忙等”,故把这种 P、V 操作称为“忙等”P、V 操作。

2.计数型信号量

计数型信号量中每个信号量至少记录两个信息:信号量的值和等待该信号量的进程队列,分别表示系统中可用的该类临界资源数量的整型变量及指向等待该类资源的进程的 PCB 链表。它的类型定义如下(用类 C 语言表述):

```
typedef struct semaphore {
   int value;//系统中空闲的该类临界资源的个数
   PCB * queue;//等待该类资源的进程链表
}s;
```

s. value≥0 时,s. queue 为空;

s. value<0 时,s. value 的绝对值为 s. queue 中等待进程的个数;

相应的 P(S)和 V(S)操作可以描述为:

```
P(S):
   s.value = s.value -1;
```

东软电子出版社

```
  if s.value < 0 then then block s.queue;
V(S):
  s.value = s.value + 1;
  if s.value≤ 0 then wakeup  s.queue;
```

P(S)和 V(S)操作使用流程图表示如图 4.12 和图 4.13 所示：

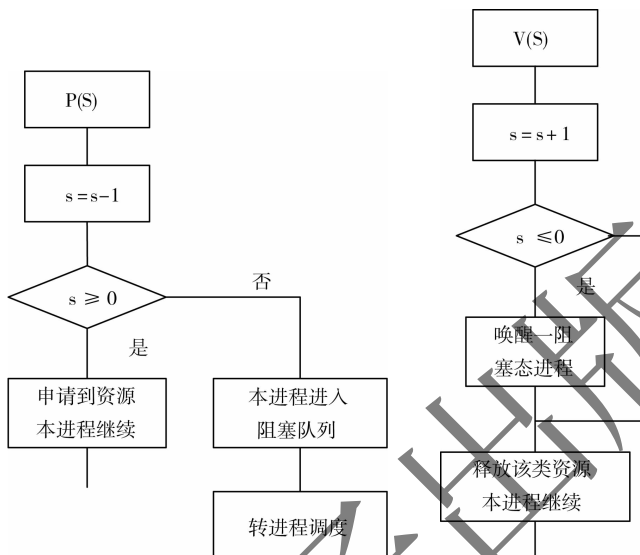

图 4.12　wait 操作原语

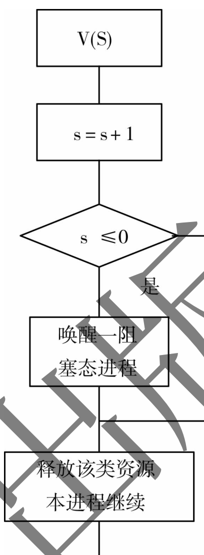

图 4.13　signal 操作原语

当进程执行 P 操作后，若 s.value < 0，则由 block 原语将它置为阻塞状态，并排在信号量链表中，然后重新调度。可见，执行 P 操作后的进程并没有陷入“忙等”，而是“让权等待”。它等待着其他进程执行 V 操作。一旦其他进程执行了 s.value + 1 操作后，发现 s.value ≤ 0，将调用 wakeup 原语，把 s.queue 链表中的第一个进程置为就绪状态后送就绪队列。

从物理概念上讲，s.value > 0 的数值表示某类可用资源的数量。每次的 P 操作，意味着请求分配一个单位的资源，因此，描述为 s.value = s.value -1。当 s.value ≤ 0 时，表示已无资源，因此请求该资源的进程将被阻塞。此时 s.value 的绝对值等于信号量链表中的等待进程数。执行一次 V 操作，意味着释放一个单位资源，因而作 s.value = s.value + 1 操作。若 s.value ≤ 0，表示链表 s.queue 中仍有因请求该资源而被阻塞的进程，因此应把 s.queue 中的第一个进程唤醒，将它转移到就绪队列中。

4.3.4　信号量集机制

1.AND 型信号量集

在某些应用条件下，一个进程需要获得两个或更多的共享资源。AND 型信号量的基本思想是将进程在整个运行过程中需要的所有资源，一次性全部分配给进程，待进程使用完后再一起释放。只要有一个资源尚未分配给进程，其他所有可能分配的资源也不能分配给他。所以，在 P 操作中增加了一个“AND”条件，故称为 AND 型信号量。在该信号量集的 P(S)和 V(S)操作中增加了“AND”条件，故称为 AND 同步，或称为同时(Simultaneous)P、V 操作 SP、SV，其定义如下：

```
SP(S1, S2,… ,Sn):
```

东软电子出版社

```
if  (S1≥ 1 && … && Sn ≥ 1 )
then
for (i = 1; i < n ;i++ )
  Si = Si -1
else
  place the process in the waiting queue associated with the first Si found with Si < 1, and set the
program count of this process to the beginning of SP operation
SV(S1, S2,… ,Sn):
for (i = 1; i < n ;i++ )
  Si = Si + 1
Remove all the process waiting in the queue associated with Si into the ready queue
```

2.一般“信号量集”

在上述的几种信号量机制中,P 和 V 操作仅能对信号量施加增 1、减 1 操作,每次只能获得或释放一个单位量的临界资源。当进程需要 N 个某类资源时,就要进行 N 次 wait 操作,这时系统的效率降低。

一般信号量集机制的基本思想是在 AND 型信号量集的基础上进行扩充,进程对信号量的 S_i 的测试值为 t_i(用于信号量的判断,即 $S_i \leqslant t_i$ 表示资源数量低于 t_i 时不分配),占用值为 d_i(用于信号量的增减,即 $S_i = S_i - d_i$ 和 $S_i = S_i + d_i$)。其 SP、SV 操作原语表示如下:

```
SP(S1,t1,d1,…,Sn,tn,dn):
  if (S1≥ t1 && … && Sn ≥ tn)
  then
  for (i = 1; i < n ;i++ )
    Si = Si - di
else
    Place the executing process in the waiting queue of the first Si with Si < ti and set its
program counter to the beginning of the SP operation
SV(S1,d1,…,Sn,dn):
  for (i = 1; i < n ;i++ )
    Si = Si - di
Remove all the process waiting in the queue associated with Si into the ready
```

下面讨论一般“信号量集”的集中特殊情况:

(1)wait(S,d,d)。此时信号量集已退化为一般信号量,表示每次申请 d 个资源,当少于 d 个时,便不分配。

(2)wait(S,1,1)。此时信号量集退化为计数信号量(S>1 时)或互斥信号量(S=1 时)。

(3)wait(S,1,0)。作为一个可控开关(当 S≥1 时允许多个进程进入临界区;当 S=0 是禁止任何进程进入临界区)。

4.4 进程间通信

每个进程各自有不同的用户地址空间,任何一个进程的全局变量在另一个进程中都看不

东软电子出版社

到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲区，进程 1 把数据从用户空间拷到内核缓冲区，进程 2 再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信。

完成本节的学习后，应能独立完成单元项目利用共享内存实现读者写者问题。

4.4.1　进程间通信的方式

相互协作的进程间需要交换一定数量的信息。虽然信号量机制作为同步工具卓有成效，但作为通信工具不够理想，因为其效率甚低，因此称为低级通信方式。而高级通信方式将以较高的效率传送大批数据。

实现进程间通信（IPC-Inter Process Communication）有如下三种基本方式：共享存储器方式、消息通信方式和共享文件方式。

1.共享存储器方式

相互通信的进程通过共享某些数据结构或存储区来进行通信，可分为共享数据结构方式和共享存储区方式。

2.消息通信方式

进程间的消息交换以消息或报文为单位，程序员利用一组通信命令（原语）来实现通信，可分为直接通信方式和间接通信方式。

（1）直接通信方式通过发送进程和接收进程来显示提供目标进程的标识符。系统通常提供两条通信原语用于发送和接收。

```
send(receiver, message);
receive(sender,message);
```

其中 receiver、sender 是消息接收者、发送者的进程号，message 是要接收或者发送的消息。直接通信方式以内存缓冲区为基础，故有较高的通信速度，它已成为单处理机多道程序系统中最流行的一种通信方式。

（2）间接通信方式是进程之间通过中间实体（如信箱）来暂存发送进程发送给某个或某些目标进程的消息，接收进程则从中取出发送给自己的消息。每个信箱由信箱头和包括若干个信格的信箱体所组成，每个信箱必须有自己的唯一标识符。利用信箱进行通信，用户可以不必写出接收进程标识符，从而也就可以向不知名的进程发送消息，且信息可以安全地保存在信箱中，允许目标用户随时读取。这种通信方式被广泛地用于多机系统和计算机网络中。

3.共享文件通信方式

利用共享文件实现进程间的通信。例如，在 UNIX 系统中，利用一个打开的共享文件来连接两个相互通信的进程，该共享文件称为管道（Pipe），因而这种通信方式又称为管道通信。为了协调双方通信，管道通信必须提供三方面的协调能力：互斥、同步、对方是否存在。

4.4.2　进程地址空间

Linux 操作系统把进程虚拟空间分为内核区和用户区两部分。内核区是指 3G-4G 的地址空间范围，操作系统内核的代码和数据等被映射到该区域；用户区是指 0G-3G 的地址空间范围，用户进程的代码和数据等被映射到该区域。在以下的章节中主要讨论用户进程的虚拟地址空间。虚拟空间的划分如图 4.14 所示。

东软电子出版社

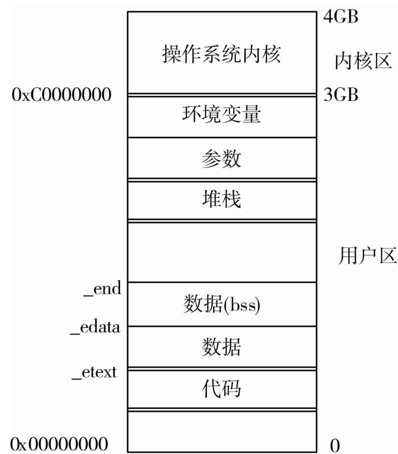

图 4.14　虚拟空间的划分

一个进程所需的虚拟空间中的各个部分未必连续，这通常会形成若干离散的虚拟“区间”(VM area)。虚拟“区间”是进程虚拟空间的一部分，这部分的虚拟空间是连续的并且有相同的一些属性。

进程地址空间是指进程能够使用的地址范围。每一个进程看到的是一个不同的线性地址(虚拟地址通过段机制转换成线性地址)；一个进程使用的地址与另一个进程使用的地址无关。内核会通过增加或删除线性地址中的区域，动态修改进程地址空间。

内核使用内存描述符结构体(mm_struct)表示进程的地址空间，该结构体包含了和进程地址空间有关的全部信息。进程地址空间由每个进程的线性地址区(vm_area_struct)组成。通过内核，进程可以给自己的地址空间动态的添加或减少线性区域。

1.内存描述符

内存描述符结构 mm_struct 的定义如下：

代码 4-1　　　　节自 include/linux/sched.h 文件

```
struct mm_struct {
        //mmap 和 mm_rb 虽然是两个不同的结构，但是却包含相同的内容，即进程地址空间中的线性地址。
只是 mmap 使用链式存储方式，而 mm_rb 使用红黑树的存储方式。
        struct vm_area_struct * mmap;          /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache; //进程最后一次引用线性区的描述符地址.
        ……
        pgd_t * pgd;   //页全局目录
        atomic_t mm_users;   //共享该内存描述符的进程个数
        atomic_t mm_count;   //内存描述符的主使用计数器
        ……
        struct list_head mmlist; //内存描述符组成的双向链表
        ……
};
```

mmap 和 mm_rb 这两个不同的数据结构描述的对象是相同的：该地址空间中的全部线性存储区域。mmap 结构体作为链表，利于简单，高效地遍历所有元素；而 mm_rb 结构体作为红-

华软电子出版社

黑树，更适合搜索指定元素。

mm_users 域记录正在使用该内存描述符的进程数目。例如，如果两个进程共享该内存描述符，那么 mm_users 的值便等于 2；mm_count 域是 mm_struct 的结构体的主引用计数，只要 mm_users 不为 0，那么 mm_count 值就等于 1。当 mm_users 值减为 0（两个线程都退出）时，mm_count 域的值才变为 0。如果 mm_count 的值等于 0，说明已经没有任何指向该 mm_struct 结构体的引用了，这时该结构体会被销毁。在进程退出调用 exit_mm() 函数过程中，首先做一些常规清除工作，更新一些内核全局统计数据。接着调用 mmput()，减少内存描述符的 mm_users 域，如果 mm_users 域变成 0，就调用 mmdrop() 函数减 mm_count 域；如果 mm_count 域变成 0，就由 free_mm() 宏调用 kmem_cache_free() 函数把 mm_struct 返还给 mm_cachp 指向的 slab 缓存。

所有的 mm_struct 结构体通过自身的 mmlist 域连接在一个双向链表中，该链表的首元素是 init_mm 内存描述符，它代表 0 号进程的地址空间。在进程的进程描述符中，mm 域存放着该进程使用的内存描述符，active_mm 指向进程运行时所使用的内存描述符。copy_process 函数利用 copy_mm 函数复制父进程的内存描述符。而像 vfork 和 clone 系统调用指定了 CLONE_VM 标志的，仅仅需要在调用 copy_mm() 函数中将 mm 域指向其父进程的内存描述符即可。而 fork 系统调用产生的子进程中的 mm_struct 结构体实际是通过文件 kernel/fork.c 中的 alloc_mm()，从 mm_cachep slab 缓存中分配得到的。

通常，每个进程都有一个唯一的 mm_struct 结构体，即唯一的进程地址空间。

2.线性区描述符

线性描述符描述地址空间内连续区间上的一个独立内存范围，它的开始和结束都必须以 4KB 方式对齐。进程只能访问某个有效的线性区，如果进程试图访问一个有效的线性区之外的地址或者用不正确的方式访问一个有效的线性区，内核将通过段异常（segmentation fault）杀死这个进程。

线性区描述符有 vm_area_struct 结构来描述，它的定义如下：

代码 4-2　　节自 include/linux/sched.h

```
struct vm_area_struct {
        struct mm_struct * vm_mm;        /* The address space we belong to. */
        unsigned long vm_start;          /* Our start address within vm_mm. */
        unsigned long vm_end;
        struct vm_area_struct *vm_next;  //指向下一个 vm_area_struct 结构体，所有的 vm_area_
struct 结构体连接成一个单向链表。
        pgprot_t vm_page_prot;           /* Access permissions of this VMA. */
        unsigned long vm_flags;          /* Flags, listed below. */
        ……
        struct rb_node vm_rb;
        ……
        struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
        struct anon_vma *anon_vma;       /* Serialized by page_table_lock */
        struct vm_operations_struct * vm_ops; //指向该结构体的操作函数的指针
        ……
};
```

东软电子出版社

vm_start 域指向线性区域的首地址,vm_end 域指向尾地址之后的第一个字节,也就是说,vm_start 是线性区域的开始地址,vm_end 是线性区域的结束地址。vm_mm 域指向和 VMA 相关的 mm_struct 结构体,注意每个 VMA 对其相关的 mm_struct 结构体来说都是唯一的,所以即使两个独立的进程将同一个文件映射到各自的地址空间,他们分别都会有一个 vm_area_struct 结构体来标志自己的内存区域;但是如果两个线程共享一个地址空间,那么他们也同时共享其中的所有 vm_area_struct 结构体。

4.4.3 Linux 进程间通信

Linux 下的进程通信手段基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD(加州大学伯克利分校的伯克利软件发布中心)在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充,形成了"system V IPC",通信进程局限在单个计算机内;后者则跳过了该限制,形成了基于套接口(socket)的进程间通信机制。Linux 则把两者继承了下来,如图 4.15 所示。

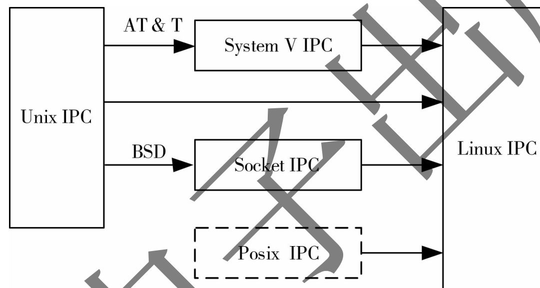

图 4.15 Linux 所继承的进程间通信

其中,最初 UNIX IPC 包括:管道、FIFO、信号;System V IPC 包括:System V 消息队列、System V 信号灯、System V 共享内存区;Posix IPC 包括: Posix 消息队列、Posix 信号灯、Posix 共享内存区。这里有有两点需要简单说明:

(1)由于 UNIX 版本的多样性,电子电气工程协会(IEEE)开发了一个独立的 UNIX 标准,这个新的 ANSI UNIX 标准被称为计算机环境的可移植性操作系统界面(PSOIX)。现有大部分 Unix 和流行版本都是遵循 POSIX 标准的,而 Linux 从一开始就遵循 POSIX 标准。

(2)BSD 并不是没有涉足单机内的进程间通信(socket 本身就可以用于单机内的进程间通信)。事实上,很多 UNIX 版本的单机 IPC 留有 BSD 的痕迹,如 4.4BSD 支持的匿名内存映射、4.3+BSD 对可靠信号语义的实现等。

图 4.15 表示出了 Linux 所支持的各种 IPC 手段,在本文接下来的讨论中,为了避免概念上的混淆,在尽可能少提及 UNIX 的各个版本的情况下,所有问题的讨论最终都会归结到 Linux 环境下的进程间通信上来。对于 Linux 所支持通信手段的不同实现版本(如对于共享内存来说,有 Posix 共享内存区以及 System V 共享内存区两个实现版本),将主要介绍 System V API,在 4.5 节将介绍 Posix 的信号量机制。

至此,可以总结出 Linux 下进程间通信的几种主要方式:

(1)管道(Pipe)及有名管道(named pipe):管道可用于具有亲缘关系进程间的通信,有名管道克服了管道没有名字的限制,因此,除具有管道所具有的功能外,它还允许无亲缘关系进程间

东软电子出版社

的通信。

(2)信号(Signal):信号是比较复杂的通信方式,用于通知接受进程有某种事件发生,除了用于进程间通信外,进程还可以发送信号给进程本身;Linux 除了支持 UNIX 早期信号语义函数 sigal 外,还支持语义符合 Posix.1 标准的信号函数 sigaction(实际上,该函数是基于 BSD 的,BSD 为了实现可靠信号机制,又能够统一对外接口,用 sigaction 函数重新实现了 signal 函数)。

(3)报文(Message)队列(消息队列):消息队列是消息的链接表,包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息,被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少,管道只能承载无格式字节流以及缓冲区大小受限等缺点。

(4)共享内存:使得多个进程可以访问同一块内存空间,是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其他通信机制,如信号量结合使用,来达到进程间的同步与互斥的目的。

(5)信号量(semaphore):主要作为进程间以及同一进程不同线程之间的同步手段。

(6)套接口(Socket):更为一般的进程间通信机制,可用于不同机器之间的进程间通信。起初是由 UNIX 系统的 BSD 分支开发出来的,但现在一般可以移植到其他类 Unix 系统上:Linux 和 System V 的变种都支持套接字。

本节的余下部分将着重讲述 Linux 的共享内存与信号量机制。

4.4.4　System V 共享内存实现

在 Linux 系统中共享内存实际上是一段特殊的内存区域,这一区域可以被两个或两个以上的进程映射到自身的地址空间中。一个进程写入共享内存中的信息,可以被其他使用这个共享内存的进程,通过一个简单的内存读操作读出来实现进程间的通信。

任何 Linux 进程创建时,都有很大的虚拟地址空间,这块虚拟地址空间只有一部分存放代码、数据、栈和堆,剩余的空间在初始时是空闲的。一块共享内存一旦被连接,即会被映射到空闲的虚拟地址空间内。随后,进程即可像对待普通的内存区域那样读、写共享内存。

进程向系统提出获得或创建共享内存的申请的时候,内核会为每一个新的共享内存段提供一个 shmid_kernel 的数据结构来维护共享内存段和文件系统之间的关系。shmid_kernel 结构位于 include/Linux/shm.h 文件中,该结构的定义如下:

代码 4-3　　　　**节自 include/linux/shm.h**

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;  //访问权限信息
        struct file *           shm_file;     //指向虚拟文件系统的指针
        unsigned long           shm_nattch; //关联到该共享内存段的进程数
        unsigned long           shm_segsz;  //共享内存段的大小
    //以下是访问时间的相关信息
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
```

华教电子出版社

```
        pid_t                   shm_cprid;  //创建者的 PID
        pid_t                   shm_lprid;  //最后访问进程的 PID
        struct user_struct      * mlock_user;  //锁定在共享内存 RAM 中的用户的 user_struct 描
述符的指针
    };
```

该数据结构中最重要的部分就是 shm_file 这个字段,它指向了共享内存对应的文件。在 struct file 结构中的字段 f_mapping 指向了该内存段使用的页面(物理内存)。同时,struct file 结构中的字段 f_path,用于指向文件系统中的文件(dentry->inode)(参见第八章的相关内容),这样就在物理内存和文件系统之间建立起了联系。当进程需要创建或者获取共享内存的时候,会先向虚拟内存系统申请各自的虚拟地址空间(用结构 struct vm_area_struct 表示),该结构中有一个成员 vm_file,它指向的就是 struct file(shm_file)。这样虚拟内存、共享内存(文件系统)和物理内存就建立了连接。共享内存各结构间的关系如图 4.16 所示。

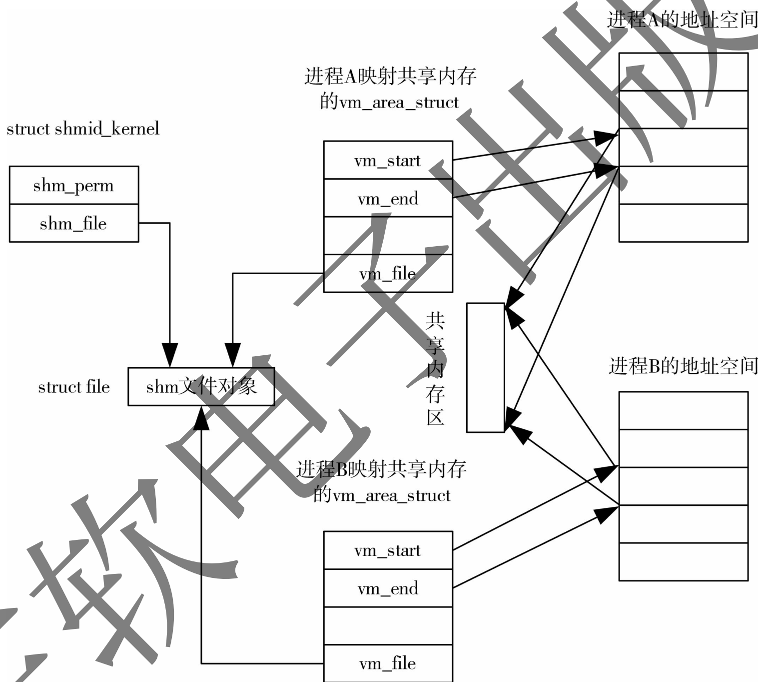

图 4.16 共享内存数据结构

东软电子出版社

对于共享内存的创建和获取主要是通过 do_shmat 来实现的,下面主要对这个函数进行分析。

代码 4-4 **节自 ipc/shm.c**

```
long do_shmat(int shmid, char __user * shmaddr, int shmflg, ulong * raddr)
{
struct shmid_kernel * shp;
……
        shp = shm_lock_check(ns, shmid); //该函数根据共享内存的 id,在内核中查找相应的 shmid
_kernel
        if (IS_ERR(shp)) {
                err = PTR_ERR(shp);
```

```
                goto out;
            }
        ……
            if (ipcperms(&shp->shm_perm, acc_mode))   //访问权限检查
                goto out_unlock;
        ……
    //以下代码用于获取共享内存对应的文件信息
    path.dentry = dget(shp->shm_file->f_path.dentry);   //获取文件表项
            path.mnt      = shp->shm_file->f_path.mnt;          //获取挂接点信息
            shp->shm_nattch++;                                   //共享内存引用计数加 1
            size = i_size_read(path.dentry->d_inode);            //通过 i 节点获取文件大小
            ……
            err = -ENOMEM;
            sfd = kzalloc(sizeof( * sfd), GFP_KERNEL);   //从 slab 中分配 shm_file_data 数据结构
(sfd)
            if (! sfd)
                goto out_put_dentry;
            file = alloc_file(path.mnt, path.dentry, f_mode, &shm_file_operations);   //分配 struct
file 结构
            if (! file)
                goto out_free;
            //以下语句实现对 file 的初始化
            file->private_data = sfd;
            file->f_mapping = shp->shm_file->f_mapping;
            sfd->id = shp->shm_perm.id;
            sfd->ns = get_ipc_ns(ns);
            sfd->file = shp->shm_file;
            sfd->vm_ops = NULL;
        ……
            user_addr = do_mmap (file, addr, size, prot, flags, 0);   //完成内存映射工作
            ……
    }
```

东软电子出版社

4.4.5　System V 共享内存编程接口

Linux 对共享内存管理的系统调用函数如下：

1.共享内存创建

在使用共享内存之前，需要先创建它。创建共享内存的系统调用函数为 shmget()，其声明如下：

```
int shmget(key_t key,int size,int shmflg);
```

其中参数 key，用于标识共享内存的键值，在调用前应赋初值；size 是以字节为单位的共享内存的大小(单位为字节)；shmflg 是将分配的共享内存属性标志，包括 IPC_CREATE(如果不

存在就创建)、IPC_EXEC(如果存在则返回失败)、IPC_NOWAIT(不等待直接返回)以及 SHM_R(可读)、SHM_W(可写)。

2.共享内存映射

在进程使用一段共享内存空间之前,需要将该共享内存与当前进程建立联系,即将共享内存映射(挂接)到当前进程。系统调用 shmat()函数将一块已存在的共享内存映像到一个进程的数据段中,该函数的声明如下:

```
void * shmat(int shmid,char * shmaddr,int shmflg);
```

其中参数,shmid 是将要连接的共享内存的标识符,即 shmget 函数的返回值;shmaddr 指定共享内存的映射地址,如果该值为非零,则将用此值作为映射共享内存的地址,如果该值为0,则由系统来选择映射的地址,一般将此值设置为 0;shmflg 用来指定共享内存的访问权限和映射条件。

3.共享内存控制

Linux 系统使用 shmctl 函数来实现共享内存空间的控制,包括读取状态、设置状态和删除操作,该函数的申明如下:

```
int shmctl(intshmid,int cmd,struct shmid_ds * buf);
```

其中参数,shmid 是共享内存的标识符,该值一般是 shmget 函数的返回值;cmd 是欲进行的操作,包括 IPC_RMID(删除)、IPC_SET(设置 ipc_perm 参数)、IPC_STAT(获取 ipc_perm 参数)、IPC_INFO(获取系统信息);buf 是缓存区域,该值随第二个参数的不同而改变。struct shmid_ds 是一个新创建的共享内存的结构表示。

4.共享内存分离

在使用完毕共享内存后,使用 shmdt 函数调用将其与当前进程分离,该函数的声明如下:

```
void * shmdt(char * shmaddr);
```

其中参数,shmaddr 是将要断开连接的共享内存的虚地址。

4.4.6 System V 信号量机制

Linux 中信号量是通过内核提供的一系列数据结构实现的,这些数据结构存在于内核空间,下面先给出信号量的数据结构(存在于 include/linux/sem.h 中)。

1.信号量的数据结构(sem)

```
struct sem {
  int  semval;          /* 信号量的当前值 */
  int  sempid;          /* 在信号量上最后一次操作的进程识别号 */
};
```

可见 Linux 使用的信号量与计数型信号量相似。信号量值用来标识系统可用资源的个数。例如可以使用信号量来标识一个缓冲区可用空间的大小(假定缓冲区大小为 256 个字节),在没有使用之前,该缓冲区没有任何内容,可用资源为 256,即 semval 值可以初始化为 256,每向缓冲区写入一个字符,信号量的值自动减 1,当信号量的值为 0 时即表示缓冲区满,资源暂不可用;每从缓冲区中读出一个字符,信号量的值自动加 1,如果信号量的值为 256,则表示缓冲区中没有内容,不可读。

2.信号量集合的数据结构(semid_ds)

```
struct semid_ds {
```

华软电子出版社

```
    struct ipc_perm sem_perm;  /* IPC 权限 */
  long  sem_otime;          /* 最后一次对信号量操作(semop)的时间 */
   long  sem_ctime;         /* 对这个结构最后一次修改的时间 */
    struct sem *sem_base;     /* 在信号量数组中指向第一个信号量的指针 */
  struct sem_queue *sem_pending;   /* 待处理的挂起操作 */
    struct sem_queue **sem_pending_last; /* 最后一个挂起操作 */
    struct sem_undo *undo;     /* 在这个数组上的 undo 请求 */
   ushort  sem_nsems;  /* 在信号量数组上的信号量号 */
  };
```

可见,Linux在管理信号量是以信号量集合的方式来进行管理。这与实际应用相一致,因为在实际应用中,两个进程间通信可能会使用多个信号量。

3.信号量集合的队列结构(sem_queue)

```
  struct sem_queue {
    struct sem_queue *next;/* 队列中下一个节点 */
    struct sem_queue **prev;  /* 队列中前一个节点, *(q->prev) == q */
    struct wait_queue *sleeper; /* 正在睡眠的进程 */
    struct sem_undo *undo;/* undo 结构 */
    int  pid;               /* 请求进程的进程识别号 */
    int  status;            /* 操作的完成状态 */
    struct semid_ds *sma;   /* 有操作的信号量集合数组 */
    struct sembuf *sops;  /* 挂起操作的数组 */
    int nsops;              /* 操作的个数 */
  };
```

Linux讨论的是信号量集合问题,对信号量集的操作与对共享内存的操作相似,通常通过以下系统调用来实现:

1.信号量集的创建

在使用信号量之前,首先要使用semget函数创建信号量集合,该函数的声明如下:

```
int semget ( key_t key, int nsems, int semflg );
```

返回值:如果成功,则返回信号量集合的IPC识别号;

如果失败,则返回-1。

该函数的第一个参数是键值,这个键值要与已有的键值进行比较,已有的键值指在内核中已存在的其他信号量集合的键值。对信号量集合的打开或存取操作依赖于semflg参数的取值:

IPC_CREAT:如果内核中没有新创建的信号量集合,则创建它;

IPC_EXCL:当与IPC_CREAT一起使用时,但信号量集合已经存在,则创建失败。

如果IPC_CREAT单独使用,semget()为一个新创建的集合返回标识号,或者返回具有相同键值的已存在集合的标识号。如果IPC_EXCL与IPC_CREAT一起使用,要么创建一个新的集合,要么对已存在的集合返回-1。IPC_EXCL单独是没有用的,当与IPC_CREAT结合起来使用时,可以保证新创建信号量集的打开和存取。

参数nsems指的是在新创建的集合中信号量的个数。

东软电子出版社

2.信号量集的控制

在 Linux 操作系统中,可以使用 semctl 函数对信号量集合及信号量集合中的信号量进行操作,该函数的原型如下:

```
int semctl ( intsemid, int semnum, int cmd, union semun arg );
```

返回值:成功返回正数,出错返回-1。

该函数的第一个参数(semid)是集合的标识号,第二个参数(semnn)是将要操作的信号量个数,从本质上说,它是集合的一个索引,对于集合上的第一个信号量,则该值为 0。

cmd 参数表示在集合上执行的命令,这些命令及解释如表 4-1 所示。

表 4.1 cmd 命令及解释

命令	解 释
IPC_STAT	从信号量集合上检索 semid_ds 结构,并存到 semun 联合体参数的成员 buf 的地址中
IPC_SET	设置一个信号量集合的 semid_ds 结构中 ipc_perm 域的值,并从 semun 的 buf 中取出值
IPC_RMID	从内核中删除信号量集合
GETALL	从信号量集合中获得所有信号量的值,并把其整数值存到 semun 联合体成员的一个指针数组中
GETNCNT	返回当前等待资源的进程个数
GETPID	返回最后一个执行系统调用 semop()进程的 PID
GETVAL	返回信号量集合内单个信号量的值
GETZCNT	返回当前等待 100%资源利用的进程个数
SETALL	与 GETALL 正好相反
SETVAL	用联合体中 val 成员的值设置信号量集合中单个信号量的值

arg 参数的类型为 semun,这个特殊的联合体在 include/linux/sem.h 中声明,对它的描述如下:

```
/* arg for semctl system calls. */
union semun{
    int val;                    /* value for SETVAL */
    struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
    ushort *array;              /* array for GETALL & SETALL */
    struct seminfo *__buf;      /* buffer for IPC_INFO */
    void *__pad;
};
```

这个联合体中,有三个成员已经在表 4-1 中提到,剩下的两个成员_buf 和_pad 用在内核中信号量的实现代码,开发者很少用到。事实上,这两个成员是 Linux 操作系统所特有的,在 UINX 中没有。

3.信号量操作

除了可以使用 semctl 系统调用访问信号量外,还可以通过 semop 函数来操作单个信号量,

东软电子出版社

该函数的声明如下：

```
int semop ( int semid, struct sembuf * sops, unsigned nsops);
```

返回值：如果所有的操作都执行，则成功返回 0；

如果出错，则为-1。

该函数的第一个参数(semid)是信号量集的识别号(可以由 semget)系统调用得到)。第二个参数(sops)是一个指针，它指向在集合上执行操作的数组。而第三个参数(nsop)是在那个数组上操作的个数。

sops 参数指向类型为 sembuf 的一个数组，这个结构在/inclide/linux/sem.h 中声明，是内核中的一个数据结构，描述如下：

```
struct sembuf {
    ushort   sem_num;        /* 在数组中信号量的索引值 */
    short    sem_op;         /* 信号量操作值(正数、负数或 0) */
    short    sem_flg;        /* 操作标志，为 IPC_NOWAIT 或 SEM_UNDO */
};
```

如果 sem_op 为负数，那么就从信号量的值中减去 sem_op 的绝对值，这意味着进程要获取资源，这些资源是由信号量控制或监控来存取的；如果没有指定 IPC_NOWAIT，那么调用进程睡眠到请求的资源数得到满足(其他的进程可能释放一些资源)。

如果 sem_op 是正数，把它的值加到信号量，这意味着把资源归还给应用程序的集合。

最后，如果 sem_op 为 0，那么调用进程将睡眠到信号量的值也为 0，这相当于一个信号量到达了 100%的利用。

综上所述，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时，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的话，系统调用返回并指明信号量上的操作没有成功，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。所以，Linux 内核在堆栈中建立一个 sem_queue 结构并填充该结构。新的 sem_queue 结构添加到集合的等待队列中(利用 sem_pending 和 sem_pending_last 指针)。当前进程放入 sem_queue 结构的等待队列中(sleeper)后调用调度程序选择其他的进程运行。

4.5　死锁

计算机系统中，资源是有限的，并且存在许多独占资源(如打印机)，一个进程在其生命期内往往需要申请多种资源。资源共享和进程的并发执行必然导致对有限资源的竞争。当进程 A 申请一个资源被进程 B 占用，而进程 B 要申请的资源又被进程 A 占有，若无外力作用则进程 A 和 B 将永远不能向前推进。这种因竞争资源从而导致两个或多个处于等待状态的进程永远不能改变其现有状态的现象称为死锁(DeadLock)。

系统发生死锁不仅浪费大量的系统资源，甚至导致整个系统崩溃。因此死锁是现代操作系统必须解决的问题，这一节将介绍死锁的概念，分析其产生的原因，并提出如何检测死锁及如何

华软电子出版社

避免死锁。

完成本节的学习后,应能独立完成单元项目哲学家就餐问题。

4.5.1 死锁产生的原因和必要条件

1.原因

(1)竞争资源:多个进程所共享的资源不足,引起进程对资源的竞争而产生死锁。

若系统中只有一台打印机 T1 和一台读卡机 R1,可供进程 P1 和 P2 共享。P1 已占有资源 T1、P2 已占有资源 R1 使用,当 P1、P2 在不释放资源 T1、R1 而又同时分别申请 R1、T1(如图 4.17 所示),形成环路,这样就会产生死锁。

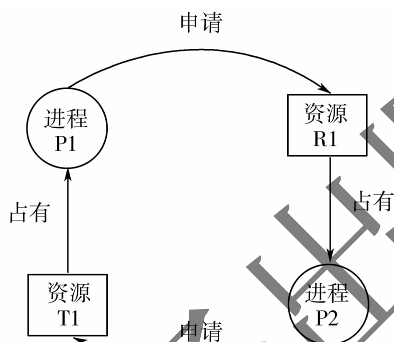

图 4.17 因竞争资源而产生死锁

(2)进程的推进顺序不当:进程运行过程中,请求和释放资源的顺序不当,而导致进程死锁。

如图 4.17 中,如果此时先让进程 P1 运行,待进程 P1 获得它所需要的两个资源 R1 和 R2 后,马上让 P1 开始运行,这样就可以避免死锁的发生。

2.必要条件

(1)互斥条件。进程要求对所分配的资源进行排它性控制,即在一段时间内资源仅能被一个进程所使用。

(2)请求和保持条件。当进程因请求资源而阻塞时,对已获得的资源保持不放。

(3)不可剥夺条件。进程已获得的资源,在未使用完之前不能被剥夺,只能在使用完是由自己释放。

(4)环路等待。在发生死锁时,必然存在一个进程——资源的环形链。

4.5.2 解决死锁的基本方法

为保证系统的正常运行,应预防或避免死锁的发生,在系统出现死锁时能检测到并解除死锁。目前处理死锁问题的方法可以归纳为如下 4 个方面:

1.预防死锁

通过设计某些限制条件,破坏产生死锁的四个必要条件中的一个或几个来防止死锁的发生。主要方法有:

(1)资源一次性分配——破坏请求和保持条件。

(2)当某进程新的资源不能满足时,释放已占有的资源——破坏不可剥夺条件。

东软电子出版社

(3)资源有序分配法,即系统给每类资源赋予一个编号,每一个进程按编号递增的顺序请求资源,释放则相反——破坏环路等待条件。

2.避免死锁

在分配资源时,不限制进程有关申请资源的命令,而是对进程的每个申请资源的活动加以动态预测。如果预测资源是安全的(即不会发生死锁),则分配资源;若不安全(即有发生死锁的可能),则不分配资源,避免死锁的典型方法是银行家算法。

银行家算法的描述如下:

一个银行家把他的固定资金贷给若干顾客,只要不出现银行家所剩的所有资金借给某个顾客还不够时,银行家的资金就是安全的。

假定顾客分成若干次进行贷款,并在第一次贷款时说明他的最大借款额。其具体算法如下:

(1)顾客的贷款依次顺序进行,直到全部操作完成。

(2)银行家对当前顾客的贷款操作进行判断,以确定其安全性,看能否支持顾客贷款,即该客户能否运行完成。

(3)安全,贷款;否则,暂不贷款。

例如:有 3 个顾客 C1、C2、C3 向银行家贷款。该银行家的资金总额为 10 个资金单位,其中客户 C1 要借 9 个资金单位,客户 C2 要借 3 个资金单位,客户 C3 要借 8 个资金单位,总计需借 20 个资金单位。若 T_0 时刻,客户占用和还需资金的状态如表 4-2 所示,则银行家应如何分配资金才是安全的呢?

表 4.2　T_0 时刻银行家的资金情况

客户	已分配资源	还需申请资源
C1	2	7
C2	2	1
C3	4	4

银行家手里剩余的资金单位是 2,此时银行家只有将资金分配给 C2 才能最终收回贷款。然后,再将资金依次分配给 C3 和 C1,这样银行家能最终收回所有贷款。

3.检测死锁

系统设有专门机构在死锁发生能够检测到死锁,并能确定参与死锁的进程及相关资源。

死锁检测的方法可以描述如下:

(1)为每个进程和每个资源指定一个唯一的号码;

(2)建立资源分配表和进程等待表,例如:

表 4.3　资源分配表

资源号	占用进程号
1	P2
2	P4
3	P1

表 4.4　进程等待表

进程号	等待资源号
P1	1
P2	2
P4	

东软电子出版社

(3)检测算法：

如上例，当进程 P4 申请资源 3 时，检测算法如图 4.18 所示。

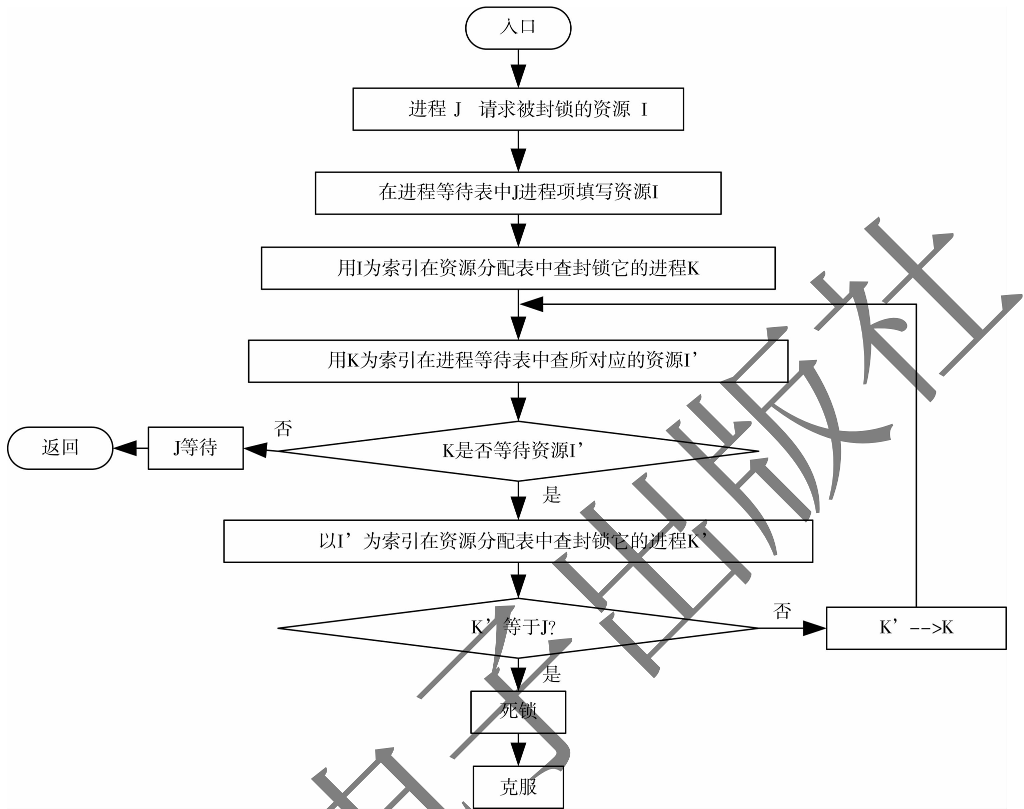

图 4.18　申请资源时的解决流程

东软电子出版社

4.解除死锁

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

(1)剥夺资源：从其他进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

(2)撤消进程：可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态消除为止；所谓代价是指优先级、运行代价、进程的重要性和价值等。

(3)进程回退：让参与死锁的进程回退到没有发生死锁前的某一点处，并由此点处继续执行，以求再次执行时不再发生死锁。

4.6　Linux 进程的实现

4.6.1　Linux 进程描述符

在 Linux 内核中，进程控制块(PCB，即进程描述符)使用 task_struct 结构来表示。该结构含有大量的信息，包含进程的基本信息、管理信息及控制信息等。请读者参考 clude/linux/sched.h 文件自行阅读，本文在使用到该结构信息时再进行说明。

4.6.2　Linux 进程的状态

进程的状态根据目的不同分别记录在进程描述符 task_struct 结构的成员变量 state、exit_state 中。Linux 系统中进程的状态由一系列的宏定义来表示，如代码 4-5 所示。

代码 4-5　　　　　　　　　　**节自 include/linux/sched.h**

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL           128
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE     (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED     (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED      (TASK_WAKEKILL | __TASK_TRACED)
```

各状态的含义如下：

1. TASK_RUNNING

表示进程正在运行或进程获得了除处理器之外的所有资源，一旦得到处理器便可立即运行，即该状态包括了正在运行状态和就绪状态的进程。在 Linux 系统中处于就绪状态的进程通常组织成一个或多个就绪队列，使用数据结构 runqueue_t 来表示。

2. TASK_INTERRUPTIBLE

表示进程处于阻塞状态(也成为浅睡眠状态)，系统当前暂时不能满足进程需要的某种资源，即进程尚不具备运行条件，即使 CPU 空闲也无法运行。处于该状态的进程睡眠在相应资源的等待队列上，当该类资源再次有效时，系统会唤醒等待队列上的进程，将其状态设置为 TASK_RUNNING。

3. TASK_UNINTERRUPTIBLE

与 TASK_INTERRUPTIBLE 状态类似，处于该状态的进程也处于阻塞状态(也成为深度睡眠状态)。不同之处在于：处于该状态的进程会一直等待直到所有资源有效或者等待超时由系统唤醒；处于该状态的进程不能响应内核投递过来的信号(signal)，而处于 TASK_INTERRUPTIBLE 状态的进程可以立即响应。

4. TASK_STOPPED

表示进程处于暂停状态，通过任务控制信号 SIGSTOP 可以将处于 TASK_RUNNING 状态的进程转换为此状态。而处于此状态的进程在接收到 SIGCONT 信号后转换到 TASK_RUNNING 状态。

电子工业出版社

5.TASK_TRACED

本来不是进程的状态,用于从停止的进程中,将当前被调试的那些进程(使用 ptrace 机制)与常规的进程区分开来。

6.TASK_DEAD

这是为了与 EXIT_DEAD 相区别的一种状态,当一个进程退出(调用 do_exit())时,将其进程描述符的 state 字段置为 TASK_DEAD。

7.TASK_KILLABLE

这是为了与 TASK_INTERRUPTIBLE 、TASK_UNINTERRUPTIBLE 相区别的一种新的睡眠状态,它的运行原理类似于 TASK_UNINTERRUPTIBLE,只不过可以响应致命信号。

以上状态是进程描述符字段 state 中的内容,对于 exit_state 字段,有以下两种状态:

1.EXIT_ZOMBIE

表示进程已经结束但尚未销毁的僵死状态,处于该状态的进程只能转换到 EXIT_DEAD 状态。处于该状态的进程已经释放了除进程描述符之外的大部分资源,等待父进程通过系统调用 wait4()来获取该进程的资源使用信息,用于系统的记账。在父进程获得该信息后,进程描述符就可以被销毁。

2.EXIT_DEAD

表示父进程已经获得了该进程的记账信息,该进程可以被销毁了。

各状态之间的转换关系如图 4.19 所示。

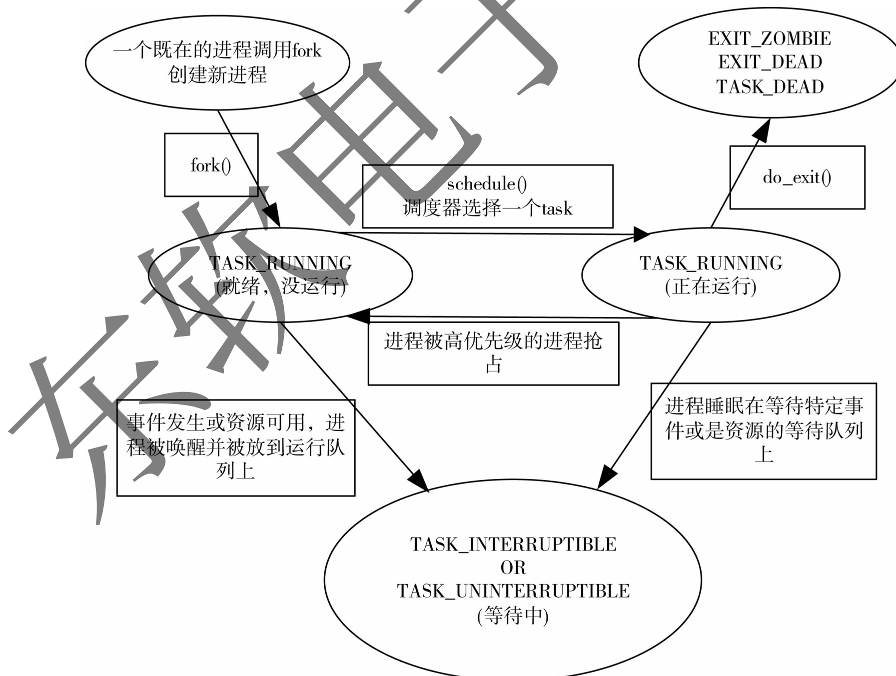

图 4.19 Linux 进程状态转换图

4.6.3 Linux 进程的虚拟内存布局

在 32 位的 x86 机器上,Linux 虚拟内存的大小为 4G。内核将这 4G 的空间分为两部分,其

东软电子出版社

中最高的 1G(从虚拟地址 0xC0000000 到 0xFFFFFFFF)空间供内核使用,称为“内核空间”。而较低的 3G 字节(从虚地址 0x00000000 到 0xBFFFFFFF),供各个进程使用,称为“用户空间”。因为每个进程可以通过系统调用进入内核,因此,Linux 内核空间由系统内的所有进程共享。

4.6.4　Linux 进程的内核堆栈

每个进程都有自己的内核栈。当进程从用户态进入内核态时,CPU 就自动设置该进程的内核栈。自然的通过寄存器%esp 即可以访问内核态栈中的元素。为进行相关的操作和设置,系统通常需要获得当前进程的描述符。如何才能够快速的获得当前进程的描述符呢?Linux 系统采用了如下的方式来解决。

内核首先将当前进程的地址以及需要快速访问的其他状态标记记录在数据结构 sturct thread_info 中,然后将该数据结构保存到内核栈栈空间的最低地址(即栈顶)处。该数据结构在内核态中的位置由数据结构 union thread_union 决定,该结构的定义如下:

代码 4-6　　　　**节自 include/linux/sched.h**

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

其中 THREAD_SIZE 的大小在通常情况下为 8192,即内核栈的大小为 8KB。通过上面的分析,可以归纳出内核栈、数据结构 struct thread_info 以及进程描述符之间的关系如图 4.20 所示。

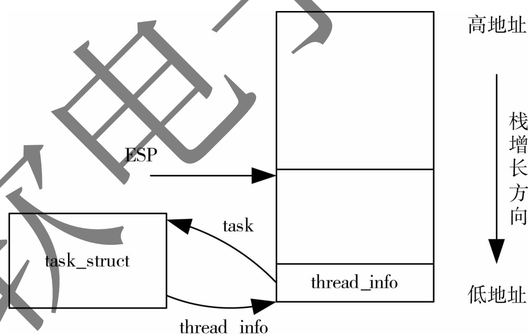

图 4.20　内核态、进程描述符与数据结构 thread_info 的关系

4.6.5　Linux 的 init 进程

在 Linux 系统中,所有进程都是由 init 进程派生而来的。init 进程的进程描述符由 init_task 静态生成,它的定义如下所示。

代码 4-7　　　　**节自 arch/x86/kernel/init_task.c**

```
struct task_struct init_task = INIT_TASK(init_task);
```

代码 4-8　　　　**节自 include/linux/init_task.h**

```
#define INIT_TASK(tsk)                          \
{                                               \
        .state = 0,                             \
        .stack = &init_thread_info,             \
```

东软电子出版社

```
        .usage = ATOMIC_INIT(2),              \
        .flags = 0,                           \
        .lock_depth = -1,                     \
        .prio = MAX_PRIO-20,                  \
        .static_prio   = MAX_PRIO-20,\
        ……
}
```

可见 init_task 的各种进程资源对象都是通过 INIT_XXX 进行初始化的,在 start_kernel()的最后由 rest_init()函数调用 kernel_thread()函数,以及 swapper 进程为“模板”建立了 kernel_init 内核进程,之后这个进程会建立 init 进程,执行/sbin//-init 文件,从而把启动过程传递到用户态。而 swapper 进程则执行 cpu_idle()函数让出 CPU,以后如果没有任何就绪的进程可调度执行,就会调度 swapper 进程,执行 cpu_idle()函数。

4.6.7 Linux 用户进程的创建

在 Linux 系统中,创建一个进程的常用方法是通过系统调用 fork 来创建一个当前进程的拷贝。fork 调用用于将调用该系统调用的进程分叉为两个内容几乎完全一样的进程,这两个进程被系统独立地进行调度、独立地竞争系统资源。通常将调用 fork 的进程成为父进程,另一个进程成为子进程。

在父进程请求 fork 系统调用、内核响应并进行处理的过程中,处于内核态的处理函数负责将父进程的进程描述符复制一份作为子进程的进程描述符。在处理完毕,即 fork 调用返回时,系统有两个几乎一模一样的进程,且这两个进程都处于将要返回到用户态的状态,也就是说 fork 系统调用是“调用一次,返回两次”。

除了 fork 系统调用外,Linux 还提供了 vfork、clone 系统调用来创建一个新进程。这几个系统调用对应的处理函数分别为 sys_fork(struct pt_regs)、sys_vfork(struct pt_regs)、sys_cloe(struct pt_regs),这些函数定义在 arch/x86/kernel/process_32.c 文件中。在内核中都调用了 do_fork 函数,该函数的主要代码如下:

代码 4-9　　　　节自 include/linux/init_task.h

```
long do_fork(unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs * regs,
            unsigned long stack_size,
            int __user * parent_tidptr,
            int __user * child_tidptr)
{
……
p = copy_process(clone_flags, stack_start, regs, stack_size,
                child_tidptr, NULL);
    if (! IS_ERR(p)) {
        struct completion vfork;
        nr = task_pid_vnr(p);
        if (clone_flags & CLONE_PARENT_SETTID)
```

华软电子出版社

```
            put_user(nr, parent_tidptr);
    ……
    }
```

可见,该函数首先调用 copy_process()函数来创建子进程描述符以及子进程执行所需要的所有其他其他内核数据结构;之后调用 wake_up_new_task()函数父子进程的调度顺序。

4.7　线程

进程是拥有资源的独立单位,也是一个可以独立调度和分派的基本单位,这是进程的两个基本特征。进程作为资源的拥有者,在进程的创建、撤销、切换过程中必然要进行资源的分配和回收、进程上下文的切换等,系统为此要付出较大的时间、空间开销。为了减少这些开销,就要求进程切换的频率不宜太高,这样系统内进程的数量就受到了限制,也就限制了程序并发执行程度的提高,从而影响了操作系统性能的提高。

如何能使多个程序更好地并发执行,同时又不至于过多地增加系统开销,已成为现代操作系统设计的目标,那么能否在不切换进程上下文的情况下提高程序的并发性呢? 这就引入了线程的概念。

完成本节的学习后,应能独立完成单元项目司机与售票员问题。

4.7.1　线程的基本概念

线程(Thread)是进程内的基本调度单位,也称为轻量级进程(light weight processes),它是比进程更小的独立运行的基本单位。在多线程操作系统中,一个进程通常包括多个线程,即意味着一个程序内的多条语句同时执行。各个线程均可作为独立调度和分派 CPU 的基本单位,它仅拥有少量必不可少的系统资源(如寄存器等),但它继承其所属进程的所有资源,这样线程切换时仅涉及少量寄存器、堆栈等。所以线程切换速度是系统开销最小的执行实体。一个传统(或重量级(heavy weight))进程只有一个控制线程。如果进程有多个控制线程,那么它能同时做多个任务。图 4.21 说明了传统的单线程进程和现代的多线程进程之间的区别。

1.线程与进程的区别

(1)拥有资源:不论是引入了线程的操作系统,还是传统的操作系统,进程都是拥有系统资源的一个独立单位,它可以拥有自己的资源。

一般地说,线程自己不拥有系统资源(除部分必不可少的资源,如栈和寄存器),但它可以访问其隶属进程的资源。即一个进程的代码段、数据段以及系统资源(如已打开的文件、I/O 设备等),可供同一进程的其他所有线程共享。

(2)调度:在传统的操作系统中,CPU 调度和分派的基本单位是进程。而在引入线程的操作系统中,则把线程作为 CPU 调度和分派的基本单位,进程则作为资源拥有的基本单位,从而使传统进程的两个属性分开,线程便能轻装运行,这样可以显著地提高系统的并发性。同一进程中线程的切换不会引起进程切换,从而避免了昂贵的系统调用。但是在由一个进程中的线程切换到另一进程中的线程时,依然会引起进程切换。

东软电子出版社

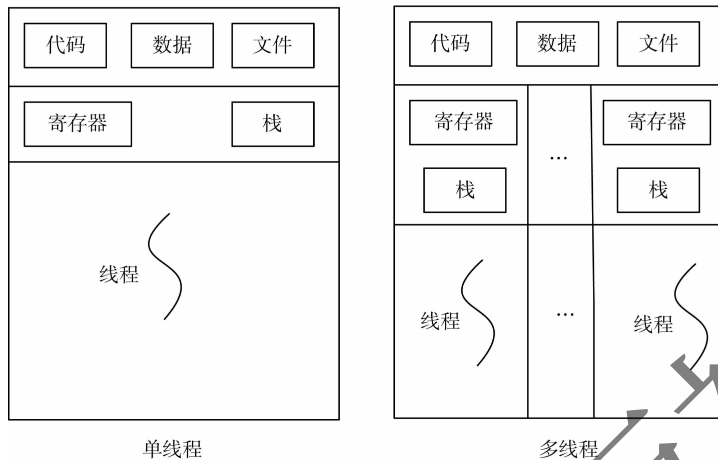

图 4.21 单线程线程和多线程线程

(3)系统开销:由于在创建或撤消进程时,系统都要为之分配或回收资源,如内存空间、I/O设备等。因此,操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地,在进行进程切换时,涉及到整个当前进程 CPU 环境的保存环境的设置以及新被调度运行的进程的 CPU 环境的设置。

而线程切换只需保存和设置少量寄存器的内容,并不涉及存储器管理方面的操作。可见,进程切换的开销也远大于线程切换的开销。此外,由于同一进程中的多个线程具有相同的地址空间,致使它们之间的同步和通信的实现也变得比较容易。在有的系统中,线程的切换、同步和通信都无需操作系统内核的干预。

(4)并发性:在引入线程的操作系统中,不仅进程之间可以并发执行,而且在一个进程中的多个线程之间也可以并发执行,因而使操作系统具有更好的并发性,从而能更有效地使用系统资源和提高系统的吞吐量。

例如,在一个未引入线程的单 CPU 操作系统中,若仅设置一个文件服务进程,当它由于某种原因被封锁时,便没有其他的文件服务进程来提供服务。

在引入了线程的操作系统中,可以在一个文件服务进程中设置多个服务线程。当第一个线程等待时,文件服务进程中的第二个线程可以继续运行;当第二个线程封锁时,第三个线程可以继续执行,从而显著地提高了文件服务的质量以及系统的吞吐量。

2.线程的状态

和进程一样,线程的主要状态有运行、就绪和阻塞。一般来说,挂起状态对线程没有什么意义。线程状态变化由相关的有五种基本操作来实现:

(1)产生(Spawn):当产生一个新进程时,同时也为该进程产生一个线程,进程中的线程可以在同一个进程中产生另一个线程,同时线程也可以产生一个新的线程。新建的线程都必须提供指令指针和参数。

(2)阻塞(Block):当线程需要等待一个事件时,它将自己阻塞,此时处理器转而执行另一个就绪线程。

(3)激活(Unlock):当阻塞一个线程的事件发生时,该线程被转移到就绪队列中。

(4)调度(schedule):选择一个就绪线程进入执行状态。

东软电子出版社

(5)结束(Finish):当一个线程完成时,其寄存器上下文和栈都被释放。

4.7.2　用户级线程与内核级线程

有两种不同方法用来提供线程支持:用户态的“用户级线程”和内核态的“内核级线程”。

1.用户级线程

建立在内核之上,并在用户态通过线程库来实现。线程库提供对线程的创建、调度和管理,无需内核的支持,因而内核并不知道用户级线程的状态,所有用户级线程的创建和调度都是在用户空间内进行的,无需进行用户态与内核态的切换操作。这样用户级线程通常能快速地创建和管理;但是用户级线程也存在缺点,例如,如果内核是单线程的,那么任何一个用户级线程若执行阻塞系统调用都会引起整个进程的所有线程被阻塞,即使其他线程都可以在应用程序内运行。

2.内核级线程

由操作系统直接支持,在内核空间内执行线程的创建、调度和管理。因为线程管理是由操作系统完成的,所以内核线程的常见和管理通常要慢于用户线程的创建和管理。但是,由于内核管理线程,当一个线程执行阻塞系统调用时,内核能调度应用程序内的另一个线程执行;而且在多处理器环境下,内核能在不同处理器上调度线程。

4.7.3　多线程模型

许多系统都提供对用户和内核级线程的支持,从而有不同的多线程模型。归纳起来主要有如下三种模型:

1.多对一模型

多对一模型是指多个用户级线程对应一个内核级线程,即对线程的管理是在用户空间中进行的。在不支持内核级线程的操作系统上所实现的用户级线程库就使用了多对一模型。

2.一对一模型

一对一模型是指一个用户级线程就对应一个内核级线程。该模型在一个线程执行阻塞系统调用时,能运行另一个线程继续执行,所以它提供了比多对一模型更好的并发功能;它能更好的支持多个线程在不同处理器上并行执行的能力。这种模型的唯一缺点是创建一个用户线程就需要创建一个相应的内核线程。由于创建内核线程的开销会影响应用程序的性能,所以这种模型的绝大多数实现限制了系统所支持的线程数量。

3.多对多模型

多对多模型是指将多个用户级线程映射到同样数量或更小数量的内核级线程上。多对多模型克服了多对一模型、一对一模型的缺点:一方面,用户可以创建任意数量的用户级线程,并且相应内核线程能在多处理器上并行执行;另一方面,当一个线程执行阻塞系统调用时,内核能调度另一个线程来执行。

4.7.4　Linux线程技术

线程技术早在20世纪60年代就被提出,但真正应用多线程到操作系统中还是在20世纪80年代中期。现在,多线程技术已经被许多操作系统所支持,包括Windows NT/2000和

东软电子出版社

Linux。

在1999年1月发布的Linux 2.2内核中,进程是通过系统调用fork创建的,新的进程是原来进程的子进程。需要说明的是,在Linux 2.2.x中,不存在真正意义上的线程,Linux中常用的线程Pthread实际上是通过进程来模拟的。

也就是说,Linux中的线程也是通过fork创建的,是"轻"进程。Linux 2.2缺省只允许4096个进程/线程同时运行,而高端系统同时要服务上千的用户,所以这显然是一个问题。它一度是阻碍Linux进入企业级市场的一大因素。

2001年1月发布的Linux 2.4内核消除了这个限制,并且允许在系统运行中动态调整进程数上限。因此,进程数现在只受制于物理内存的多少。在高端服务器上,即使只安装了512MB内存,现在也能轻而易举地同时支持1.6万个进程。

在Linux 2.5内核中,已经做了很多改进线程性能的工作。在Linux 2.6中改进的线程模型仍然是由Ingo Molnar来完成的。它基于一个1:1的线程模型(一个内核线程对应一个用户线程),包括内核内在的对新NPTL(Native Posix Threading Library)的支持,这个新的NPTL是由Molnar和Ulrich Drepper合作开发的。

2003年12月发布的Linux 2.6内核,对进程调度经过重新编写,去掉了以前版本中效率不高的算法。进程标识号(PID)的数目也从3.2万升到10亿。内核内部的大改变之一就是Linux的线程框架被重写,以使NPTL可以运行其上。

Linux的另一种可选线程模型是IBM开发的NGPT(Next Generation Posix Threads for Linux),它是基于GNU Pth(GNU Portable Threads)项目而实现的M:N模型(M个用户态线程对应N个核心态线程)。在2003年,IBM放弃了NGTP。

NPTL的设计目标可归纳为以下几点:POSIX兼容性、SMP结构的利用、低启动开销、低链接开销(即不使用线程的程序不应当受线程库的影响)、与LinuxThreads应用的二进制兼容性、软硬件的可扩展能力、多体系结构支持、NUMA支持,以及与C++集成等。

4.7.5 Glibc中的NTPL库编程接口

像每个进程有一个进程ID一样,每个线程也有一个线程ID,进程ID在整个系统中是唯一的,但线程不同,线程ID只在它所属的进程环境中有效。线程ID用pthread_t数据类型来表示,实现的时候用一个结构来代表pthread_t数据类型,所以在可以移植的操作系统中不能把它作为整数处理。

1.线程的创建

函数原型:

```
int pthread_create(pthread_t * restrict tidp,const pthread _attr_t * restrict attr,void * ( * start_rtn)(void),void * restrict arg);
```

返回值:若成功返回则返回0,否则返回错误编号。

当pthread_creat成功返回时,tidp指向的内存单元被设置为新创建线程的线程ID。attr参数用于定制各种不同的线程属性。可以把它设置为NULL,创建默认的线程属性。

新创建的线程从start_rtn函数的地址开始运行,该函数只有一个无类型指针参数arg,如果需要向start_rtn函数传递的参数不止一个,那么需要把这些参数放到一个结构中,然后把这个结构的地址作为arg参数传入。

华软电子出版社

2.线程的终止

线程是依进程而存在的,当进程终止时,所有的线程也就终止了。当然也有在不终止整个进程的情况下停止它的控制流(即线程)的方法。

(1)线程只是从启动进程中返回,返回值是线程的退出码。

(2)线程可以被同一进程中的其他线程取消。

(3)线程可以调用 pthread_exit 自己退出。

函数原型:

```
void pthread_exit(void * rval_ptr);
```

rval_prt 是一个无类型指针,与传给启动例程的单个参数类似。进程中的其他线程可以调用 pthread_join 函数访问到这个指针。

3.获取线程的终止状态

函数原型:

```
intpthread_join(pthread_t thread,void * * rval_ptr);
```

返回值:若成功返回 0,否则返回错误编号。

当一个线程通过调用 pthread_exit 退出或者简单地从启动历程中返回时,进程中的其他线程可以通过调用 pthread_join 函数获得进程的退出状态。调用 pthread_join 进程将一直阻塞,直到指定的线程 thread 调用 pthread_exit 从启动例程中终止或者被取消。

如果线程是从它的启动历程返回,rval_ptr 将包含返回码。

4.7.6　POSIX 信号量

POSIX 信号量是一个 sem_t 类型的变量,它有两种信号量的实现机制:无名信号量和命名信号量。无名信号量可以用在共享内存的情况下,例如实现进程中各个线程之间的互斥和同步;命名信号量通常用于不共享内存的情况下,例如不共享内存的进程之间。

1.POSIX 无名信号量编程接口

(1)sem_init 函数。

在使用信号量之前,必须对信号量进行初始化。sem_init 函数初始化指定的信号量,它的原型为:

```
int sem_init(sem_t * sem, int pshared, unsigned value);
```

参数 sem 指向要初始化的信号量,参数 value 为信号量的初始值,参数 pshared 用于说明信号量的共享范围,如果 pshared 为 0,那么该信号量只能由初始化这个信号量的进程中的线程使用,如果 pshared 非零,任何可以访问到这个信号量的进程都可以使用这个信号量。

返回值,如果初始化成功,sem_init 返回 0,如果不成功,sem_init 返回-1 并设置 errno 的值。如表 4.5 所示,列出了 sem_init 可能发生的错误和对应得错误码。

表 4.5　sem_init 可能发生的错误和对应得错误码

错误	原因
EINVAL	value 大于 SEM_VALUE_MAX
ENOSPC	初始化资源已经耗尽,或者信号量的数目超出了 SEM_NSEMS_MAX 的范围
EPERM	调用程序没有适当的特权

东软电子出版社

(2)sem_destroy 函数。

函数 sem_destroy 销毁一个指定的信号量,它的原型为:

```
int sem_destroy(sem_t *sem);
```

参数 sem 为指向要销毁的信号量的指针。

返回值,如果销毁成功,sem_destroy 返回 0,如果不成功,sem_destroy 返回-1 并设置 errno 值。如果 sem 不是有效的信号量,sem_destroy 就将 errno 置为 EINVAL 。

(3)sem_post 函数。

sem_post 函数实现对指定信号量的 signal 操作,它的原型为:

```
int sem_post(sem_t *sem);
```

参数 sem 为要进行 signal 操作的信号量的指针。

返回值,如果操作成功,sem_post 返回 0。如果不成功,sem_post 返回-1,并设置 errno 值。如果 sem 不是有效的信号量,sem_post 就将 errno 置为 EINVAL 。

(4)sem_wait 函数。

sem_wait 函数实现对指定信号量的 wait 操作。sem_trywait 函数与 sem_wait 类似,只是在试图对一个为零的信号量进行操作时,它不会阻塞调用线程,而是立即返回。这两个函数的原型为:

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

参数 sem 为要进行 wait 操作的信号量的指针。

如果操作成功,这两个函数返回 0,如果不成功,这些函数返回-1 并设置 errno。如果 sem 不是有效的信号量,sem_ wait 就将 errno 置为 EINVAL。如果 sem_trywait 在信号量为零时执行 wait 操作,则将 errno 置为 EAGAIN 。

sem_post,sem_wait 和 sem_trywait 同样可用于命名信号量。

2.POSIX 命名信号量编程接口

之所以称为命名信号量,是因为它有一个名字、一个用户 ID、一个组 ID 和权限,这些是提供给不共享内存的那些进程使用命名信号量的接口。命名信号量的名字是一个遵守路径名构造规则的字符串。

(1) sem_open 函数。

sem_open 函数用于创建或打开一个命名信号量。它的原型为:

```
sem_t *sem_open(const char *name, int oflag);
```

参数 name 是一个标识信号量的字符串。参数 oflag 用来确定是创建信号量还是连接已有信号量。如果设置了 oflag 的 O_CREAT 比特位,则会创建一个新的信号量。

(2) sem_close 函数。

sem_close 函数用于关闭命名信号量。它的原型为:

```
int sem_close(sem_t *sem);
```

参数 sem 是指向要关闭的信号量的指针。单个程序可以用 sem_close 函数关闭命名信号量,但是这样做并不能将信号量从系统中删除,因为命名信号量在单个程序的执行之外是具有持久性的。当进程调用_exit、exit、exec 或从 main 返回时,进程打开的命名信号量同样会被关闭。

返回值,如果成功,sem_ close 返回 0,如果不成功,sem_ close 返回-1 并设置 errno 值。如

东软电子出版社

果 sem 不是有效的信号量,sem_ close 就将 errno 置为 EINVAL 。

(3)sem_unlink 函数。

sem_unlink 函数用于在所有进程关闭了命名信号量之后,将信号量从系统中删除。它的原型为:

```
int sem_unlink(const char * name);
```

参数 name 为要删除的命名信号量的名字。如果成功,sem_ unlink 返回 0,如果不成功,sem_ unlink 返回-1 并设置 errno 值。表 4.6 列出了 sem_ unlink 可能发生的错误和对应得错误码。

表 4.6　sem_ unlink 可能发生的错误和对应得错误码

错误	原因
EACCES	权限不正确
ENAMETOOLONG	name 比 PATH_NAME 长,或者它有一个组件超出了 NAME_MAX 的范围
ENOENT	信号量不存在

4.8　单元项目

4.8.1　进程管理的模拟实现

【项目描述】

假设系统有 a、b、c 三类资源,分别记为 ra、rb、rc,各资源的可用数量均为 10。在系统中分别创建 3 个进程,ID 号分别为 1、2、3,各进程需要的 a 类资源数为 3、b 类资源数为 4、c 类资源数为 5。各进程的运行时间设定为一个随机值。创建进程并将获得了全部资源的进程组织在就绪队列中,将未能获得资源的进程组织在阻塞队列中。每个进程在运行一个时间单位后让出 CPU 的使用权重新进入就绪队列尾,而使处于就绪队列头的进程开始运行。当某进程的运行时间已满时,从就绪队列中删除。

【构思设计】

(1)数据结构设计,为模拟题目所提高的功能,对用不着的 PCB 的某些字段进行删减,设计了如下的 PCB 结构,PCB 采用链式结构的方式进行组织。

```
typedef struct pcb{
    int id;              //进程的 ID 号
    int ra;              //所需资源 A 的数量
    int rb;              //所需资源 B 的数量
    int rc;              //所需资源 C 的数量
    int ntime;           //所需的时间片个数
    int rtime;           //已经运行的时间片个数
    char state;          //进程状态 W:就绪,B:阻塞,R:运行
    struct pcb * next;   //指向下一个 PCB 的指针
}PCB;
```

同时定义两个全局 PCB 的指针变量 ready、block,分别表示就绪队列及阻塞队列。以及三个全局整型变量 ra、rb、rc,分别表示 a、b、c 三类资源的可用数量,他们的初始值都为 10,并随着

东软电子出版社

资源的分配而减少。

(2)创建初始就绪队列以及阻塞队列,即填充 PCB 各个域的值,先满足第一个 PCB 的资源需求,然后是第二个,依次类推,对资源满足要求的 PCB 将其插入就绪队列中,并将其状态设置为就绪态对不满足要求的 PCB 插入阻塞队列中。

(3)进程运行,当就绪队列中还存在 PCB 时,在就绪队列中选择第一个 PCB 来模拟运行,将其状态设置为运行态,并将其运行时间加 1,然后判断运行时间是否与所需时间相等,如果相等表示已运行完成,需将该进程终止。则先回收其占用的资源,然后将其 PCB 释放掉。对于运行时间小于所需时间的进程将其 PCB 重新插入就绪队列尾。

(4)资源回收及再分配,当终止的进程释放其所占用的资源后,依次从阻塞队列中取 PCB,如果所需资源数已达到要求则将资源分配给该进程,并将其从阻塞队列中移除插入到就绪队列中。

【实施运行】

项目流程图如图 4.22 所示。

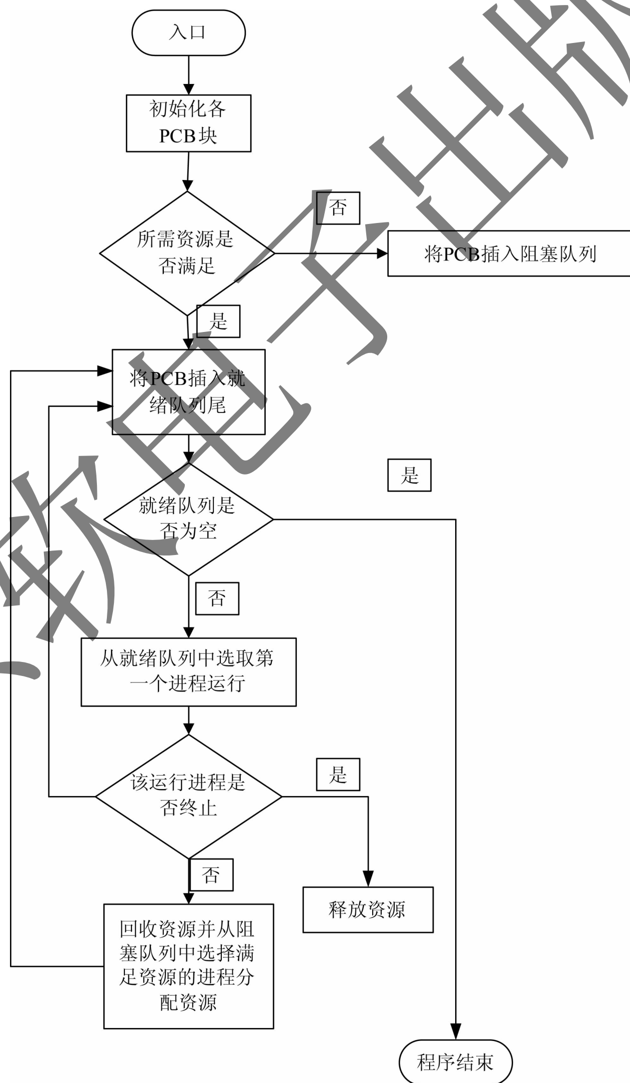

图 4.22 进程管理项目流程图

东软电子出版社

项目的运行效果图如图 4.23 和图 4.24 所示。

```
dxq@dxq-desktop: ~
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
dxq@dxq-desktop:~$ ./pcb

ready queue information*****************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
         1        W        3         4         5         2                 0
         2        W        3         4         5         5                 0

blocked queue information***************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
         3        B        3         4         5         4                 0

running process information*************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
        1         R        3         4         5         2                 1
Press enter key to continue...
```

图 4.23　项目运行效果图(1)

```
ready queue information*****************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
         1        W        3         4         5         2                 1
         2        W        3         4         5         5                 1

blocked queue information***************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
         3        B        3         4         5         4                 0

running process information*************************************************
        PID       STATU    RA        RB        RC        need Time         run Time
        1         R        3         4         5         2                 2


                PID 1's process has finished
Press enter key to continue...
```

图 4.24　项目运行效果图(2)

4.8.2　生产者—消费者问题

【项目描述】

一个仓库可以存放 K 件物品。生产者每生产一件产品,就将产品放入仓库,仓库满了就停止生产。消费者每次从仓库中取一件物品,然后进行消费,仓库空时就停止消费。

【构思设计】

生产者—消费者(Producer-Consumer)问题是最著名的进程同步问题。它描述了一组生产者向一组消费者提供消息(message),它们共享一个有界缓冲池(bounded-buffer pool),生产者向其中投放消息,消费者从中取得消息。生产者—消费者问题是许多相互合作进程的一种抽象。

在本例中,我们假定缓冲池中具有 n 个缓冲区,每个缓冲区存放一个消息,可以用互斥信号量 mutex 使各进程对缓冲池实现互斥访问;利用 empty 和 full 计数信号量分别表示空缓冲及满缓冲的数量。又假定这些生产者和消费者互相等效,只要缓冲池未满,生产者便可将消息送

东软电子出版社

入缓冲池;只要缓冲池未空,消费者便可从缓冲池取走一个消息。生产者—消费者的构思设计如下:

1.数据结构

intarray [k];整型数组用来模拟仓库,k 为常量,表示仓库的大小。

intin,out;指示存取位置的指针。in 记录第一个为空的缓冲区,out 记录第一个不为空的缓冲区。

int empty,full,mutex;empty 控制缓冲区不满,full 控制缓冲区不空,mutex 用于保护临界区;初始时 empty=k,full=0,mutex=1。

进程控制块的结构如下:

```
struct
{
    char name[10];        //进程名
    STATUS state;//状态
}
```

为简化起见,只保留了进程名及进程状态,其中进程状态由运行、就绪、等待、终止四种状态组成。STATUS 的定义为:

```
typedef enum {run,ready,wait,end}STATUS;
```

2.生产者与消费者进程

producer(生产者进程)原语操作:

```
Typeitem;
{
while (true)
{
produce(item);
p(empty);
p(mutex);
buffer[in]:=item;
in:=(in+1) mod k;
v(mutex);
v(full);
}
}
```

consumer(消费者进程)原语操作:

```
Type item;
{
while (true)
{
p(full);
p(mutex);
item:=buffer[out];
out:=(out+1) mod k;
```

东软电子出版社

```
consume(item);
v(mutex);
v(empty);
}
}
```

【实施运行】

以随机的方式启动生产者、消费者进程,生产者进程依次产生“WELCOMEOS!”的各个字符,消费者进程从缓冲区中取出。运行效果如图 4.25 所示。

```
dxq@dxq-desktop:~$ ./product-consumer
*********************consume()**************************
process name is: consumer
critial resource can not use!

*************************produce()**********************
process name is: producer
Now produce No.[0] product W

*********************consume()**************************
process name is: consumer
Now consumer No.0 product is W

*********************consume()**************************
process name is: consumer
critial resource can not use!

*********************consume()**************************
process name is: consumer
critial resource can not use!

*********************consume()**************************
process name is: consumer
critial resource can not use!

*************************produce()**********************
process name is: producer
Now produce No.[1] product E

*************************produce()**********************
process name is: producer
Now produce No.[2] product L

*********************consume()**************************
process name is: consumer
Now consumer No.1 product is E
```

图 4.25　生产者—消费者运行效果图

4.8.3　利用共享内存实现读者写者问题

【项目描述】

利用 Linux 建立两个进程,一个是 writer,一个是 reader。writer 从用户处获得输入,然后将其写入共享内存。reader 从共享内容获取信息,然后在屏幕上打印出来。

【构思设计】

利用共享内存在进程之间传递信息时,需要有一种途径让 reader 知道什么时候 writer 已经把信息放入了共享内存,这自然会联想到上节提到的同步机制。这里就介绍一下 Linux 中的信号量机制。

【实施运行】

对于 writer 首先创建信号量及共享内存(或者是读取两者的 ID 值),并挂接共享内存,然后

东软电子出版社

初始化信号量的值为0。接着进入死循环,首先读取信号量的值是否为0,如果为0,表示可以写入数据,则从键盘输入数据写入共享内存中,然后执行信号量加1操作,表示允许读操作,系统显示此时不能再写数据。如果输入的数据是“end”则表示结束通信,并卸载共享内存。

对于reader首先创建信号量或者共享内存(或者是读取两者的ID值),并挂接共享内存。接着进入死循环,然后读取信号量的值是否为1,如果为1,表示可以读入数据,将从共享内存中读出数据输出,然后执行信号量自减1操作,表示运行写入操作,系统显示此时不能再读数据。如果读取的数据是“end”则表示结束通信,将卸载共享内存,并删除共享内存及信号量

程序的运行效果如图4.26和图4.27所示。

```
dxq@dxq-desktop:~/writer-reader$ ./writer
write data operate
please input something:welcome
write data operate
please input something:to
write data operate
please input something:os
write data operate
please input something:end
dxq@dxq-desktop:~/writer-reader$
```

图4.26 writer运行效果图

```
dxq@dxq-desktop:~/writer-reader$ ./reader
read data operate
welcome
read data operate
to
read data operate
os
dxq@dxq-desktop:~/writer-reader$
```

图4.27 reader运行效果图

4.8.4 哲学家就餐问题

【项目描述】

如图4.28所示,假设有五位哲学家围坐在一张圆形餐桌旁,做以下两件事情之一:吃饭,或者思考。吃东西的时候,他们就停止思考,思考的时候也停止吃东西。餐桌中间有一大碗意大利面,每两个哲学家之间有一只餐叉。因为用一只餐叉很难吃到意大利面,所以假设哲学家必须用两只餐叉吃东西,而且他们只能使用自己左右手边的那两只餐叉。

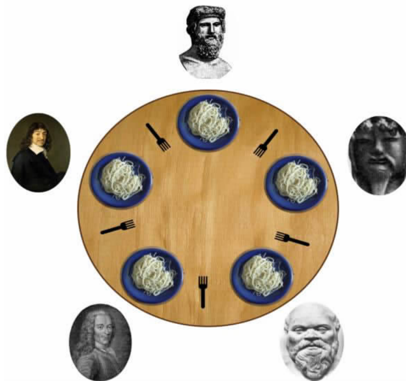

图4.28 哲学家就餐问题

华软电子出版社

【构思设计】

哲学家从来不交谈，这就很危险，可能产生死锁。每个哲学家都拿着左手的餐叉，永远都在等右边的餐叉（或者相反）。

即使没有死锁，也有可能发生资源耗尽。例如，假设规定当哲学家等待另一只餐叉超过五分钟后就放下自己手里的那一只餐叉，并且再等五分钟后进行下一次尝试。这个策略消除了死锁（系统总会进入到下一个状态），但仍然有可能发生“活锁”。如果五位哲学家在完全相同的时刻进入餐厅，并同时拿起左边的餐叉，那么这些哲学家就会等待五分钟，同时放下手中的餐叉，再等五分钟，又同时拿起这些餐叉。

在实际的计算机问题中，缺乏餐叉可以类比为缺乏共享资源。一种常用的计算机技术是资源加锁，用来保证在某个时刻，资源只能被一个程序或一段代码访问。当一个程序想要使用的资源已经被另一个程序锁定，它就等待资源解锁。当多个程序涉及到加锁的资源时，在某些情况下就有可能发生死锁。例如，某个程序需要访问两个文件，当两个这样的程序各锁了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

这里使用 Linux 的 fork 函数来创建进程，我们将讨论 fork 函数该函数的实现，并以此学习 Linux 进程是如何管理的。

【实施运行】

在本项目中，使用信号量集来实现各个哲学家的同步。利用 fork 函数产生 5 个不同的进程，每个进程模拟一个哲学家。当某个哲学家要进餐时，首先调用 p 操作锁住左边的餐叉，接着调用 p 操作锁住右边的餐叉。当哲学家获得左右两把餐叉时即可开始进餐。进程完毕后，哲学家依次调用 v 操作释放左右两边的餐叉。

程序的运行效果图如图 4.29 所示。

```
dxq@dxq-desktop:~$ ./zhexuejia
This sem number is 196611
P2 begin to eat  1 times.
P2 begin think 3 seconds.
P3 begin to eat  1 times.
P3 begin think 3 seconds.
P4 begin to eat  1 times.
P4 begin think 3 seconds.
P1 begin to eat  1 times.
P1 begin think 13 seconds.
P5 begin to eat  1 times.
P5 begin think 3 seconds.
P2 begin to eat  2 times.
P2 begin think 6 seconds.
P3 begin to eat  2 times.
```

图 4.29　哲学家进餐问题运行效果图

读者可以修改信号的同步与互斥关系，使程序产生死锁的效果。

4.8.5　司机与售票员问题

【项目描述】

采用 Linux 的线程方式实现司机与售票员间的同步与互斥，司机与售票员问题的描述参见 4.3.2 节的内容。

东软电子出版社

【构思设计】

在该例子中,使用线程来模拟司机与售票员的活动。并使用 POSIX 信号量来实现二者之间的同步。

例子中,设置 void driver(void *arg);、void conductor(void *arg);两个线程实例分别表示司机与售票员的动作序列。并设置两个 sem_t 类型的变量 sem_t semStop;、sem_t semRun;分别表示汽车停止与正常运行的信号量。

司机线程实例的动作序列如下所示。

```
void driver(void *arg)
{
    while(1)
    {
        sem_wait(&semRun);
        启动车辆
    正常行驶
    到站停车
        sem_post(&semStop);
    }
}
```

售票员线程实例的动作序列如下所示。

```
void conductor(void *arg)
{
    while(1)
    {
        sem_post(&semRun);
        售票
        sem_wait(&semStop);
        开车门
        乘客上下车
        关车门
    }
}
```

【实施运行】

由于 NTPL 库是用户库,所以在编译多线程应用程序的时候,需要添加 pthread 库。编译命名如下所示。

```
ccniit@ccniit:~$ gcc -o bus bus.c - lpthread
```

程序的运行效果图如图 4.30 所示。

东软电子出版社

```
stop the bus
open the bus door
passenger in_out
close the bus door
selling ticket
start-up bus
running
stop the bus
open the bus door
passenger in_out
close the bus door
selling ticket
start-up bus
running
stop the bus
```

图 4.30　司机与售票员程序运行效果图

4.9　本章小结

本章围绕计算机操作系统的核心任务之一——进程管理，详细讨论了进程的定义、特征、状态及其转换；进程互斥与同步、进程间通信以及死锁的问题。并以三个经典问题的实现贯穿全文，同时讨论了 Linux 进程管理的相关问题以及部分 Linux 内核 API 函数。

4.10　习题

一、填空题

1. 当一个进程独占处理器顺序执行时，具有两个特性：________________和可再现性。

2. 进程由程序、数据和________________组成。

3. 在信号量机制中，信号量 S > 0 时的值表示________________；若 S < 0，则表示________________，此时进程应________________。

4. 采用对换方式在将进程换出时，应首先选择处于________________且优先权低的进程换出内存。

5. UNIX 系统向用户提供的用于创建新进程的系统调用是________________。

6. 常用的进程通信方式有管道、________________、________________和邮箱机制。

7. 正在执行的进程等待 I/O 操作，其状态将由执行状态变为________________状态。

8. 在操作系统中的异步性主要是指________________。

9. 在生产者—消费者问题中，消费者进程的两个 wait 原语的正确顺序为：________________和________________。

10. 一次只允许一个进程访问的资源叫________________。

11. 如果信号量的当前值为 4，则表示________________，如果信号量的当前值为-4，则表示________________。

12. 当一个进程完成了特定的任务后，系统收回这个进程所占的________________和取消

该进程的________就撤消了该进程。

13. 对信号量S的操作只能通过________操作进行,对应每一个信号量设置了一个等待队列。

14. 若干个事件在同一时刻发生称为并行,若干个事件在同一时间间隔内发生称为________。

15. 现代操作系统的特征是________、________、虚拟和异步性

16. 产生死锁的四个必要条件是互斥条件和请求和保持,不剥夺条件和________。

17. 对信号量S的操作只能通过________操作进行,对应每一个信号量设置了一个等待队列。

二、选择题

1. 进程所请求的一次打印输出结束后,将使进程状态从( )

A. 运行态变为就绪态　　B. 运行态变为等待态

C. 就绪态变为运行态　　D. 等待态变为就绪态

2. 一作业进入内存后,则所属该作业的进程初始时处于( )状态。

A. 运行　　B. 等待

C. 就绪　　D. 收容

3. 共享变量是指( )访问的变量。

A. 只能被系统进程　　B. 只能被多个进程互斥

C. 只能被用户进程　　D. 可被多个进程

4. 临界区是指并发进程中访问共享变量的( )段。

A. 管理信息　　B. 信息存储

C. 数据　　D. 程序

5. 若系统中有五台绘图仪,有多个进程均需要使用两台,规定每个进程一次仅允许申请一台,则至多允许( )个进程参于竞争,而不会发生死锁。

A. 5　　B. 2

C. 3　　D. 4

6. 产生系统死锁的原因可能是由于( )。

A. 进程释放资源　　B. 一个进程进入死循环

C. 多个进程竞争,资源出现了循环等待　　D. 多个进程竞争共享型设备

7. 产生死锁的四个必要条件是互斥条件和(1),不剥夺条件和(2)。

(1)

A. 请求和阻塞条件　　B. 请求和释放条件

C. 请求和保持条件　　D. 释放和阻塞条件

E. 释放和请求条件

(2)

A. 线性增长条件　　B. 环路条件

C. 有序请求条件　　D. 无序请求条件

8. 产生死锁的基本原因是(1)和(2)。

东软电子出版社

(1)

A. 资源分配不当　　B. 系统资源不足

C. 作业调度不当　　D. 资源的独占性

(2)

A. 进程推进顺序非法　　B. 进程调度不当

C. 系统中进程太多　　D. CPU 运行太快

9. 现代操作系统的两个基本特征是(　　)和资源共享。

A. 多道程序设计　　B. 中断处理

C. 程序的并发执行　　D. 实现分时与实时处理

10. 引入多道程序的目的在于(　　)。

A. 充分利用 CPU,减少 CPU 等待时间

B. 提高实时响应速度

C. 有利于代码共享,减少主、辅存信息交换量

D. 充分利用存储器

11. 我们把在一段时间内,只允许一个进程访问的资源,称为临界资源,因此,我们可以得出下列论述,正确的论述为(　　)。

A. 对临界资源是不能实现资源共享的

B. 只要能使程序并发执行,这些并发执行的程序便可对临界资源实现共享

C. 为临界资源配上相应的设备控制块后,便能被共享

D. 对临界资源,应采取互斥访问方式,来实现共享

12. 对于记录型信号量,在执行一次 P 操作时,信号量的值应当(　　);在执行 V 操作时,信号量的值应当(　　)。

A. 不变　　B. 加 1

C. 减 1　　D. 加指定数值

E. 减指定数值

13. 产生死锁的四个必要条件是互斥条件和(　1　),不剥夺条件和(　2　)。

(1)

A:请求和阻塞条件　　B:请求和释放条件

C:请求和保持条件　　D:释放和阻塞条件

E:释放和请求条件

(2)

A. 线性增长条件　　B. 环路条件

C. 有序请求条件　　D. 无序请求条件

三、简答题

1. 在单处理机环境下,进程间有哪几种通信方式,是如何实现的?

2. 简述进程的几种状态和引起状态转换的典型原因,以及相关的操作原语。

3. 在生产者—消费者问题中,能否将生产者进程的 wait(empty)和 wait(mutex)语句互换,为什么?

东软电子出版社

4.什么是死锁？产生死锁的四个必要条件是什么？

5.在生产者—消费者问题中，如果缺少了 signal(full)或 signal(empty)，对执行结果会有什么影响？

6.这是一个从键盘输入到打印机输出的数据处理流图，其中键盘输入进程通过缓冲区 buf1 把输入数据传送给计算进程，计算进程把处理结果通过缓冲 buf2 传送给打印进程。buf1 和 buf2 为临界资源，试写出键盘输入进程，计算进程及打印进程间的同步算法。

7.设公共汽车上，司机和售票员的活动分别是：

司机:	售票员:
启动车辆	上乘客
正常行车	关车门
到站停车	售票
	开车门
	下乘客

在汽车不断地到站，停车，行使过程中，这两个活动有什么同步关系？并用 wait 和 signal 原语操作实现它们的同步。

8.设系统中有三种类型的资源(A,B,C)和五个进程(P1,P2,P3,P4,P5)，A 资源的数量为 17,B 资源的数量为 5,C 资源的数量为 20。在 T0 时刻系统状态如表 1 和表 2 所示。

系统采用银行家算法实施死锁避免策略。

a、T0 时刻是否为安全状态？若是，请给出安全序列。

b、在 T0 时刻若进程 P2 请求资源(0,3,4)，是否能实施资源分配？为什么？

c、在 b 的基础上，若进程 P4 请求资源(2,0,1)，是否能实施资源分配？为什么？

d、在 c 的基础上，若进程 P1 请求资源(0,2,0)，是否能实施资源分配？为什么？

表 1　　T0 时刻系统状态

	最大资源需求量			已分配资源数量		
	A	**B**	**C**	**A**	**B**	**C**
P1	5	5	9	2	1	2
P2	5	3	6	4	0	2
P3	4	0	11	4	0	5
P4	4	2	5	2	0	4
P5	4	2	4	3	1	4

表 2　　T0 时刻系统状态

	A	**B**	**C**
剩余资源数	2	3	3

9.系统中有五个进程 P1、P2、P3、P4、P5，有三种类型的资源：R1、R2、和 R3。在 T0 时刻系统状态如表所示。若采用银行家算法实施死锁避免策略，回答下列问题：

T0 时刻是否为安全状态？为什么？

若这时 P4 请求资源(1,2,0)，是否能实施资源分配？为什么？

东软电子出版社

在上面的基础上，若进程 P3 请求资源(0,1,0)，是否能实施资源分配？为什么？

T0 时刻系统状态

	已分配资源数量			最大资源需求量		
	R1	R2	R3	R1	R2	R3
P1	0	0	1	0	0	1
P2	2	0	0	2	7	5
P3	0	0	3	6	6	5
P4	1	1	5	4	3	5
P5	0	3	3	0	6	5

T0 时刻系统状态

	R1	R2	R3
剩余资源数	3	3	0

10. 系统运行有三个进程：输入进程、计算进程和打印进程，它们协同完成工作。输入进程和计算进程之间共用缓冲区 buffer1，计算进程和打印进程之间共用缓冲区 buffer2。输入进程接收外部数据放入 buffer1 中；计算进程从 buffer1 中取出数据进行计算，然后将结果放入 buffer2；打印进程从 buffer2 取出数据打印输出。用算法描述这三个进程的工作情况，并用 wait 和 signal 原语实现其同步操作。

11. 进程 A1，A2，…，An 通过 K 个缓冲区向进程 B1，B2，…，Bm 不断地发送消息。发送和接收工作遵循如下规则：

每个发送进程一次发送一个消息，写入缓冲区，缓冲区大小与消息长度一致；

对每个消息，B1，B2，…，Bm 都需接收一次，读入各自的数据区内；

K 个缓冲区都满时，发送进程等待，没有可读的消息时，接收进程等待。

试用 wait 和 signal 原语操作组织正确的发送和接收操作。

12. Jruassic 公园有一个恐龙博物馆和一个公园. 有 m 个旅客和 n 辆车，每辆车只能容纳一个旅客。旅客在博物馆逛了一会儿，然后排队乘坐旅行车。当一辆车可用时，它载入一个旅客，然后绕公园行驶任意长的时间。如果 n 辆车都已被旅客乘坐游玩，则想坐车的旅客需要等待；如果一辆车已经就绪，但没有旅客等待，那么这辆车等待。使用信号量同步 m 个旅客和 n 辆车的进程。

13. 读者与写者问题（reader-writer problems ）。在计算机体系中，对一个共享文件进行操作的进程可分为两类：读操作和写操作，它们分别被称为读者和写者。访问该文件时读者和写者，写者和写者间必须实现互斥。只有在没有读者访问文件时，写者才允许修改文件。或者写者在修改文件时不允许读者去读，否则会造成读出的文件内容不正确。试写出算法描述读者和写者的问题。

14. 今有三个并发进程 R，M，P，它们共享了一个可循环使用的缓冲区 B，缓冲区 B 共有 N 个单元。进程 R 负责从输入设备读信息，每读一个字符后，把它存放在缓冲区 B 的一个单元中；进程 M 负责处理读入的字符，若发现读入的字符中有空格符，则把它改成“，”；进程 P 负责

东软电子出版社

把处理后的字符取出并打印输出。当缓冲区单元中的字符被进程P取出后，则又可用来存放下一次读入的字符。请用PV操作为同步机制写出它们能正确并发执行的伪代码。

15.理发店里有一位理发师、一把理发椅子和五把供等候理发的顾客坐的椅子。如果没有顾客，理发师便在理发椅上睡觉。当一个顾客到来时，他必须先叫醒理发师，如果理发师正在理发时又有顾客来到，而如果有空椅子可坐，他们就坐下来等，如果没有空椅子，他就离开。这里的问题是为理发师和顾客各编写一段程序来描述他们行为，并用wait和signal原语操作实现其同步。

16.在原语执行期间，是否可以响应中断？为什么？

17.不同用户的不同任务之间的进程是有临界区？为什么？请举例说明。

18.用整型信号量描述在哲学家进餐问题中，至多允许4个哲学家同时进餐的算法。

东软电子出版社