第 3 章 进程控制

操作系统的目标之一是让计算机系统以高效、便捷、安全、稳定的方式执行特定任务。程序是执行这些任务的指令序列集合。程序在执行过程中，需要在操作系统的统一管理下分配执行任务所需要的资源，接收数据，并按照程序的步骤或指令的顺序对数据或资源进行逐步处理。操作系统为方便监控和管理任务的执行过程，将程序的动态执行过程抽象为进程。

3.1 程序的顺序和并发执行

程序由一组保存在存储器中的指令组成。中央处理器(CPU)执行一条指令所需的全部步骤和时间称为一个指令周期。如图 3.1 所示，处理机的一个指令周期主要包括两个阶段：取指阶段和执行阶段。取指阶段中，处理机从存储器中一次读取一条指令，而执行阶段可能涉及更多的操作，这取决于指令类型本身。当机器关机或发生未知错误，或程序正常执行完毕时，程序执行才会停止。若没有中断机制，则在程序计数器的自增机制下，处理机不停地执行取指令、执行指令操作。因此，程序的执行在时间上是严格按先后次序相继地进行操作的过程，体现了计算机完成相应功能时所应采取的顺序步骤。

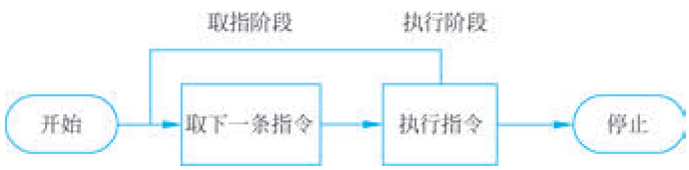

图 3.1　基本指令周期

程序的顺序执行具有如下特征。

(1) 顺序性。处理机的操作严格按照程序所规定的顺序执行，即每一操作必须在上一个操作结束之后开始。

(2) 封闭性。程序是在封闭的环境下执行的，即程序运行时独占全机资源，资源的状态只有本程序才能改变它。程序一旦开始执行，其执行结果不受外界因素的影响。

(3) 可再现性。只要程序执行时的环境和初始条件相同，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“停停走走”地执行，都将获得相同的结果。

程序顺序执行时的特性为程序开发人员调试程序和检测错误带来了极大便利。然而，若系统每次只执行一个程序，则无法充分利用处理机资源。当一个程序在申请 I/O 资源(如读取磁盘或网络数据)时，低速的 I/O 设备会导致处理机长期闲置。此时，如果多个程序或任务可同时载入计算机系统，让这些程序在计算机系统内并发执行，处理机就可以分配给其他程序。如此，可显著提高资源的利用率和处理机的响应速度，从而改善用户体验。

程序的并发执行是指一组在逻辑上相互独立的程序,或程序段在执行过程中,其执行时间在客观上允许互相重叠,即一个程序段的执行尚未结束,另一个程序段的执行已经开始的这种执行方式。程序的并发执行可以显著提高系统吞吐量。然而,若缺乏有效的并发执行控制机制,则会产生与程序顺序执行不同的特征。

(1) 间断性。程序在并发执行时,由于它们共享系统资源,以及为完成同一项任务而相互合作,这些并发执行的程序之间形成了相互制约的关系。相互制约将导致并发程序具有“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,这些资源的状态将由多个程序来改变,导致程序的运行失去了封闭性。因此某程序在执行时必然会受到其他程序的影响。一个程序的内部变量不再受到保护,其他程序可以直接或间接地修改这些内部变量,从而导致程序的不可预测行为、安全漏洞和维护困难等问题。

(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。例如,有两个循环程序 A 和 B,它们共享一个变量 N。程序 A 每执行一次时,都要做 N=N+1 操作;程序 B 每执行一次时,都要执行 print(N)操作,然后将 N 置 0。程序 A 和 B 以不同的速度运行,这样可能出现下述 3 种情况(假设某时刻变量 N 的值为 n)。

① N=N+l 在 print(N)和 N=0 之前,执行后的 N 值分别为 n+1、n+l、0。

② N=N+l 在 print(N)和 N=之后,此时得到的 N 值分别为 n、0、1。

③ N=N+1 在 print(N)和 N=0 之间,此时得到的 N 值分别为 n、n+l、0。

上述情况说明,程序在并发执行时,由于失去了封闭性,其计算结果与并发程序的执行速度有关,从而使程序的执行失去了可再现性,也就是说,程序经过多次执行后,虽然它们执行时的环境和初始条件相同,但得到的结果却各不相同。在实际系统运行中,程序的执行还涉及不同用户、不同的软硬件资源以及程序间通信等因素的影响。如果并发执行的程序段不按照特定的规则和方法进行资源共享和竞争,则必然会得到非预料或不正确的结果。因此在支持多道程序并发的操作系统中,应提供相应的机制来解决以上问题,保证程序执行的正确性和一致性。

这一问题的解决可以追溯到 1966 年判定多个语句并发执行的 Bernstein 条件。Bernstein 条件将程序中的任一语句 S_i 划分为两个变量的集合 $R(S_i)$ 和 $W(S_i)$。其中,$R(S_i)=\{a_j\}_{j=1}^{m}$ 是语句 S_i 在执行期间需要进行读的变量的集合;$W(S_i)=\{b_j\}_{j=1}^{n}$ 是语句 S_i 在执行期间需要进行修改的变量的集合。

如果语句 S_1 和 S_2 同时满足:

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

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

(3) $W(S_1)\cap W(S_2)=\varnothing$;

则语句 S_1 和 S_2 是可以并发执行的。例如,语句 S_1 为 $c=a-b$,S_2 为 $w=c+1$,则 $R(S_1)=\{a, b\}$,$W(S_1)=\{c\}$,$R(S_2)=\{c\}$,$W(S_2)=\{w\}$,此时,3 个条件均满足,因此两个语句可以并发执行。

然而,系统要判定多个并发执行的各程序段是否满足 Bernstein 条件是相当困难的。目前,在操作系统中普遍使用进程(process)这一概念描述程序的执行过程并隔离不同进程的执行状态,以保证程序执行的独立性。

3.2 进程和进程切换

3.2.1 进程的概念

进程早在20世纪60年代研发MULTICS时就被提出并实现，它是描述程序执行活动的一个动态实体，也是支持程序运行的一种系统机制，还是对处理机执行程序的动态活动的一种抽象。使用进程后，操作系统中一个程序或多个程序中的不同部分可以并发执行，提高了系统的吞吐量和响应速度。进程间的相互隔离有助于防止未授权的访问和潜在的安全风险，增强了操作系统的安全性。操作系统可以通过进程来管理任务，包括创建新任务、终止无用或有害的任务以及调整任务的优先级。通过抽象出进程概念对程序的执行过程进行隔离，这种抽象体现在以下两个层面。

（1）进程是独立的逻辑控制流。每个进程都有它自己的指令序列要执行，这构成了它的逻辑控制流。即使两个进程执行相同的程序，它们也会各自遵循独立的执行路径。进程为每个程序提供了一种假象，就好像它独占地使用处理机，处理机以进程为单位来保证整体行为与程序的顺序执行行为完全一致。为此，操作系统提供了多种并发和同步机制（如信号量、互斥锁、条件变量等），使得多个进程之间能够协调地工作，避免资源竞争和数据不一致等问题，该内容将在第4章详细介绍。

（2）进程对其所使用的各种资源进行了抽象。首先，进程为每个程序提供了一种假象，就好像它独占地使用系统地址空间，其所使用的地址实际上是虚拟地址。存储管理服务将多级硬件存储器统一管理，实现了虚拟内存这种抽象，该内容将在第6章详细介绍。其次，进程还使用了文件和I/O设备等资源。进程通过系统调用等机制与文件系统进行交互，实现文件的打开、关闭、读写等操作。这一方式使得程序能够方便地访问和管理存储在文件系统中的数据或者操作外部设备，该内容将在第7、8章的文件管理和设备管理中分别介绍。

总之，进程作为操作系统中的核心概念，提供的两大抽象分别对应处理机调度和资源分配的基本单位。这些抽象不仅使得程序能够并发地执行，还提高了系统的安全性、稳定性、共享性和灵活性。为此，进程可以定义为：进程是程序在处理机上可并发执行的一次执行过程的动态实体，是处理机独立调度和资源分配的基本单位。

有了进程概念提供的这两种抽象，从程序员角度看，程序是独占地使用处理机和内存，处理机不间断地逐条执行程序中的指令，程序的代码和数据似乎是内存中唯一的对象。从实现上看，进程就是内存中刻画运行程序状态的一种数据结构，其综合描述了系统的动态变化情况和当前资源（包括处理机、内存、设备和文件等）分配的情况。进程在执行过程中，需要在操作系统的统一管理下分配执行任务所需要的资源，接收数据，并按照程序的步骤或指令的顺序对数据或资源进行逐步计算处理。

进程和程序是两个紧密相关但完全不同的概念。两者的不同表现在如下方面。

（1）程序是静态的，本质是指令集合，仅告诉处理机如何一步步地执行，而不是真正地执行，类似于我们做计划的规划书、做饭参照的菜谱、建造房子使用的建筑设计图纸；而进程是动态的，是程序在处理机上的一次执行过程，类似于一次做饭过程、建造房子的过程等。

（2）程序是指令的有序集合，而进程是并发且异步的，进程的执行不是一贯到底的，而是走走停停的。

（3）程序可以作为软件资料长期保存在磁盘中，而进程是有生命周期的，进程终止后就不

复存在了。

例如，当双击桌面上的 WPS 图标启动 Word 时，操作系统会创建一个进程来运行这个程序。此时，可以打开文档、编辑文本、保存文件等。每次启动 WPS，都会创建一个新的进程；如果同时打开了两个不同的文档，则会有两个独立的 WPS 进程在运行。

这个示例深刻地揭示了进程与程序的紧密联系。

（1）一个程序可对应多个进程：同一程序可以运行在多个数据集上，当多次运行时，它将属于不同的进程。

（2）进程的创建离不开程序：在创建进程时，需要加载可执行程序中的程序段和数据段到内存中，从而构建进程内存映像结构。

总的来说，进程就是程序每次执行的一个动态实例，既能描述程序的并发执行，又是系统资源分配的基本单位。进程可以说是计算机系统或操作系统中最基本、最重要、最核心的概念。操作系统的所有内容都是围绕进程展开的，并为进程服务。进程的大多数特征也反映了操作系统的特征。

3.2.2　进程的内存映像结构

为执行程序中对应的指令序列，进程必须从可执行文件中加载必要的程序和数据信息。以 Intel x86 架构下的 Linux 系统为例，如图 3.2 所示，程序文件对应的初始化段(init)、代码段(text)和只读数据段(rodata)总是被加载到起始虚拟地址 0x08048000 处，而数据段(data)和未定义段(bss)总是加载到只读数据段的后面。这些段被统一称为静态区，这个区域中的所有数据均来自可执行文件。

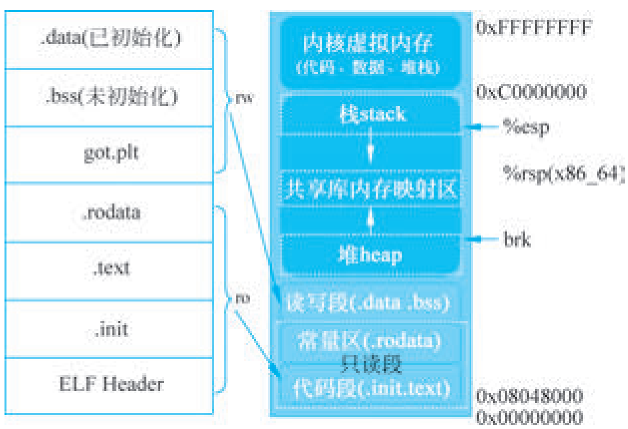

图 3.2　进程的内存映像

程序运行时的堆和栈则是动态分配的。其中，堆在程序运行时允许通过 malloc 函数动态申请内存地址空间，也叫作运行时堆，其紧挨着.bss 段并向高地址方向增长。栈主要用于支持函数调用的运行环境，是从用户空间的最高地址处向低地址方向增长，也被称为运行时用户栈。在堆和栈之间的一块空间保留给共享库的目标代码，栈以上的高地址空间保留给操作系统内核。

3.2.3　进程控制块

仅通过程序自身的各种代码段和数据段，操作系统无法充分描述其动态运行过程。虽然堆栈结构能刻画当前函数调用执行的动态过程，但这远远不够。操作系统需要通过某种数据结构记录进程当前详细状态，以支持多个并发进程的执行和资源分配的管理。因此，操作系统

为每个进程创建了一个重要的数据结构——进程控制块(Process Control Block,PCB)。PCB和代码段对其进行操作的数据段构成了进程的静态描述。PCB是进程存在的唯一标识,也是操作系统感知进程、管理进程的唯一数据结构对象。PCB中包含操作系统为每个进程创建的进程标识信息、进程在处理机现场的状态信息、进程的控制信息以及与进程调度相关的重要信息。

1. 进程标识信息

进程的标识信息主要包括进程的唯一标识号(PID)、进程所属用户的唯一标识号(UID)以及进程所属父进程的唯一标识号(PPID)等。一个进程通常有两种标识符。

(1) 内部标识符。操作系统为每一个进程赋予了一个唯一的数字标识符,它通常是一个进程的序号。设置内部标识符主要是为了方便系统管理。

(2) 外部标识符。它由创建者提供,通常由字母、数字组成,往往是由用户在访问该进程时使用。为描述进程的家族关系,还应设置父进程标识和子进程标识。此外,还可设置用户标识,以指示拥有该进程的用户。

2. 处理机状态信息

处理机状态信息主要包括处理机所有寄存器的当前取值内容。处理机在运行程序时,许多重要信息都临时加载到寄存器中,当处理机被中断时,所有信息都必须保存在PCB中,以便在该进程重新执行时,能从被中断的位置继续执行。

(1) 通用寄存器。临时保存数据的处理机内部存储单元,包括算术逻辑运算的结果、指令操作数和地址等信息。

(2) 程序计数器(Program Counter,PC)。存放要访问的下一条指令的地址。

(3) 程序状态字(Program State Words,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个与之相关的用户栈,用于存放过程和系统调用参数及调用地址,栈指针指向该栈的栈顶。

3. 进程调度信息

进程调度主要依据的是进程的优先级、进程执行时间统计、进程被阻塞的事件以及进程的类型等信息。为此,在PCB中还存放了一些与进程调度和进程切换有关的信息。

(1) 进程状态。指明进程的当前状态,作为进程调度的依据。

(2) 进程优先级。用于描述进程使用处理机的优先级别,一般用整数表示,优先级高的进程应优先获得处理机。

(3) 进程调度所需的其他信息。它们与所采用的进程调度算法有关,例如进程已等待处理机的总时间、进程已执行的总时间等。

(4) 事件。指进程由执行状态转变为阻塞状态所等待发生的事件,即阻塞原因。

4. 进程控制信息

进程控制信息包括进程当前所拥有的资源信息。资源信息不仅包括进程在内存中的内存地址相关信息,如页表基址,同时还包括进程当前正打开的文件列表以及所有占用的I/O硬件资源列表信息。

(1) 代码段和数据段的地址。指进程的机器代码和数据所在的内存或外存地址。

(2) 进程同步和通信机制。指实现进程同步和进程通信时必需的机制,如消息队列指针、信号量等,它们可能全部或部分地放在PCB中。

（3）资源清单。除处理机以外的、进程所需的全部资源和已经分配到该进程的资源的列表。

（4）链接指针。它给出了本进程PCB所在队列中的下一个进程的PCB首地址。

由于PCB掌握着进程的重要信息，会被系统频繁访问，因此所有进程的PCB对象均由操作系统创建，并常驻系统内存区中集中管理。当一个新进程被创建时，操作系统为之创建PCB实例；当进程结束时，销毁对应的PCB实例。在这个过程中，处理机调度模块、资源分配模块、中断处理例程、性能监控与分析模块都要在内核态下频繁读取和修改PCB信息。目前，常用链表和索引两种数据结构来组织管理所有进程的PCB。

（1）链表结构：将相同状态的进程PCB链接在一起，分别组成就绪队列和阻塞队列，如图3.3所示。链表结构易于实现，但当系统查找特定进程PCB时，需要遍历该PCB前序所有的PCB实例，效率较低。

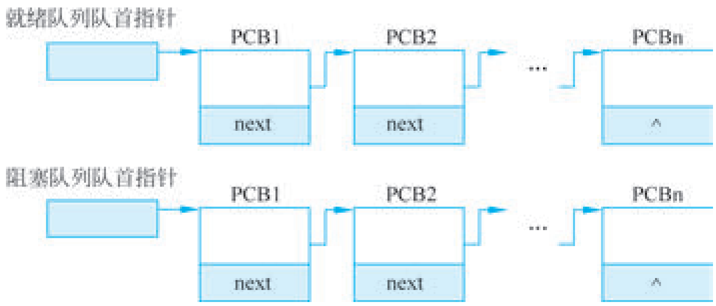

图3.3 PCB链表结构

（2）索引结构：将相同状态的进程通过索引表组织起来，索引表的每个表项指向一个PCB，如图3.4所示。该组织方式能有效提高PCB的遍历效率，同时与指针相结合，可有效提高PCB的创建和销毁效率。

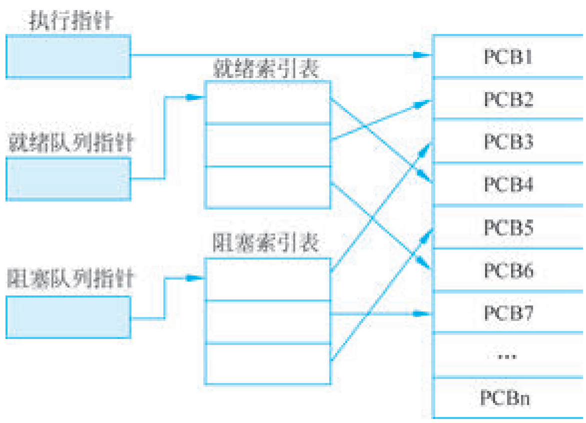

图3.4 PCB索引结构

进程执行需要访问众多系统资源，除PCB表外，操作系统还需要对所有系统资源进行统一管理。管理这些系统资源所需的数据结构主要如下。

（1）内存表，通常指页表(Page Table)，是操作系统用来实现虚拟内存到物理内存映射的数据结构。在现代计算机中，程序运行时使用的是虚拟地址空间，而实际的数据存储在物理内存中。为将虚拟地址转换为物理地址，操作系统维护了一个或多个页表。每个进程都有自己的页表，它记录了该进程所使用的每个虚拟页面与实际物理页面的映射关系。

（2）I/O 表，也称为设备表或 I/O 端口表，是用来管理硬件设备及其驱动程序之间通信的数据结构。它包含所有已连接到系统的 I/O 设备的信息，并且对于每一个设备，操作系统都会保留一个或多个表项来描述其状态和配置。

（3）文件打开列表，有时也称为文件描述符表，是操作系统用来跟踪当前进程中已打开文件的一项机制。这个列表对于管理文件访问权限、同步多个对同一文件的操作以及确保正确关闭不再需要的文件是非常重要的。

总之，从理论角度来看，进程是对正在运行的程序过程的抽象；从管理和实现角度来看，进程是一种存在于内存中的数据结构，包括 PCB、数据段、程序段、堆栈段的重要信息。

3.2.4 进程上下文切换

不同进程的并发执行本质上是多进程交替执行的过程。操作系统提供了进程切换机制以解决交替执行这一问题。进程上下文（Context）是一个与进程切换和处理机状态切换有关的概念，是进程执行过程中顺序关联的静态描述。内核为每个进程维护了一个上下文。简单地说，上下文即内核重启被中断进程所需的全部状态信息，这些信息包括通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构。

在不发生进程切换时，进程上下文的改变都是在同一进程内进行的，此时，每条指令的执行对进程上下文的改变较小，一般反映为指令寄存器、程序计数器以及保存调用子程序返回地址等的变化。提出进程上下文的概念主要是为了进程上下文的切换。进程上下文的切换发生在不同的进程之间，而不是在同一个进程内。

在进程执行过程中，由于中断、阻塞或程序出错等原因造成进程切换。这时操作系统需要存储进程当前执行位置和相关信息，并恢复另一进程的相关信息，同时从其之前被中断的位置继续运行。我们把已执行过的进程指令和数据在相关寄存器与堆栈中的内容称为上文，把正在执行的指令和数据在寄存器与堆栈中的内容称为正文，把待执行的指令和数据在寄存器与堆栈中的内容称为下文。

进程上下文切换过程一般包含 3 个步骤，并涉及 2 个进程。为展示这一过程，简单起见，假设有两个进程 P_0、P_1，处理机正在运行 P_0，如图 3.5 所示。

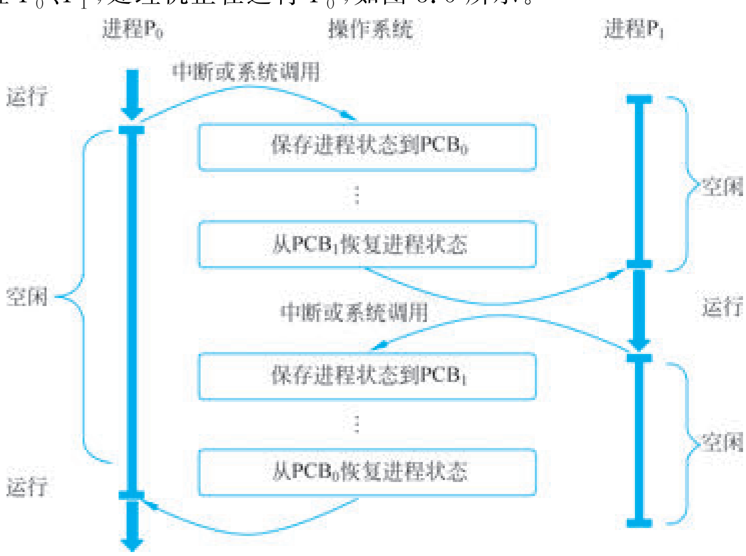

图 3.5 进程上下文切换

(1) 保存被切换进程 P_0 的正文部分(或当前状态)至 PCB_0 中。因为 P_0 的时间片已用完或因申请外部资源等原因导致 P_0 陷入内核,故操作系统会将 P_0 的程序计数器、通用寄存器、栈指针寄存器、基址寄存器、段寄存器、标志寄存器等处理机现场保存到 P_0 的 PCB_0 数据结构中。与此同时,将 P_0 的状态从"运行态"切换成"就绪态"。为提高系统执行效率,有的计算机在设计时采用了多组寄存器技术,即进程上下文切换时不把被切换进程上下文的正文保存到相关存储区,而是直接切换相应的寄存器。这样就节省了数据保存和获取所耗费的时间。

(2) 操作系统中调度模块开始执行,选取进程 P_1。进程在执行过程中,处理器中断当前进程,并选择一个之前被中断的进程或新创建的进程,这个决策过程即调度,相应功能由内核中的调度模块实现。当内核选择一个新的进程并分配给它的处理机时,我们说内核调度了该进程。相关知识将在第 5 章详细介绍。

(3) 将被选中进程 P_1 原来被保存的正文部分从 PCB_1 中恢复过来,并送至有关寄存器与堆栈中,具体包括程序计数器、通用寄存器、栈指针和基址指针、段寄存器和标志寄存器等,这些信息被恢复到处理机现场,为重新运行被选中进程 P_1 做准备。

当 P_1 执行一定时间后,若时间片用完或被阻塞等,也需要保存进程处理机状态到 PCB_1,然后由调度程序选择进程 P_0 或另一个进程,执行一段时间后因再次使用外部设备或需要进行系统调用时陷入内核,接着从 P_0 的 PCB_0 或另一个进程的 PCB 中恢复进程状态,继续执行。

简言之,上下文切换过程主要分为 3 个步骤:①保存当前被中断进程的上下文;②选择另一进程;③恢复被选中进程的上下文信息并执行。

尽管上下文切换是操作系统实现多任务处理的核心机制之一,但它也会带来一定的开销。

(1) 保存和恢复寄存器状态:每次上下文切换都需要保存和恢复大量的寄存器状态,这涉及多次内存访问操作,增加了处理机的负担。

(2) 缓存失效:当进程切换时,处理机缓存中的数据可能会无效,导致后续指令执行时需要重新加载数据,影响性能。

(3) TLB(Translation Lookaside Buffer)刷新:如果进程使用的虚拟地址空间不同,则操作系统可能需要刷新 TLB,这会导致额外的内存访问延迟,TLB 快表的相关知识将在第 6 章详细解释。

(4) 调度器开销:操作系统需要花费时间来选择下一个要执行的进程,并更新调度队列,这也增加了上下文切换的开销。

3.3 进程状态及状态转换

进程在它的生命周期中会经历多个状态的变化,这是因为进程的执行具有间断性。也就是说,进程并不是从创建到终止一直连续地占用处理机资源,而是根据系统的调度策略和自身的运行需求在不同的状态间转换。

当操作系统创建一个新进程时,该进程首先会以未运行的状态加入系统的某个队列等待被执行;从系统调度的角度看,操作系统从队列中挑选一个进程来执行,被选中的进程就会过渡到运行状态。然而,不是所有进程都做好了执行任务的准备,如果进程执行过程中所需要的资源未准备好,即使分配了处理机也无法执行,这就好比在食堂排队,好不容易轮到你,结果饭卡忘带了,在你取回饭卡前无法办理任何业务。为方便区别这两种状态,未运行状态的进程可以进一步划分为就绪态和阻塞态。

(1) 就绪态(Ready State):除处理机外,进程其他所有必需的资源都已准备好,进程就被放置到就绪队列中,处于就绪态。就绪态意味着进程已经准备好执行,只是当前没有获得处理机,它在等待调度器选择它占用处理机来执行。

(2) 阻塞态/等待态(Blocked State):进程在运行过程中申请某些资源或需要等待某个事件的发生,在资源未满足前或事件发生前,进程将从运行状态转换到阻塞态。导致进程阻塞的典型事件有请求 I/O、申请缓冲空间、用户输入、信号量等。通常操作系统将处于阻塞态的所有进程组织成阻塞队列。有的系统则根据不同的阻塞原因而把处于阻塞态的进程排成多个队列。

(3) 执行态/运行态(Running State):当进程被调度并获得处理机的执行时,进程处于执行态/运行态。单处理机系统中最多只有一个进程处于执行态,而多处理机系统允许多个进程同时处于执行态。

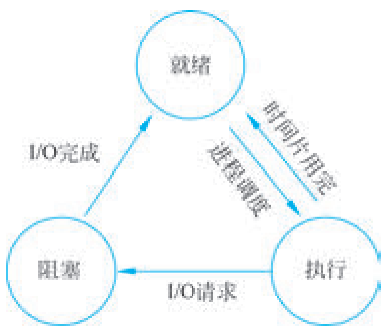

图 3.6　进程的 3 种基本状态及其转换

进程在生命周期中会经历多个状态的变化。图 3.6 给出了进程的 3 种基本状态以及各状态之间的转换关系。处于就绪态的进程,在调度程序为之分配了处理机之后,该进程便可执行,相应地,进程就由就绪态转变为执行态。正在执行的进程,当分配给它的时间片已用完而被暂停执行时,该进程便由执行态恢复到就绪态;当进程执行过程中申请 I/O 资源,因 I/O 不可用或正在准备中导致进程执行受阻而无法继续执行,则该进程将由执行态转变为阻塞态;阻塞态的进程等待的事件完成后,该进程将由阻塞态转变为就绪态,等待再次被调度分配处理机。

在实际系统中,为了管理的需要,还存在两种比较常见的进程状态,即创建态和终止态。

(4) 创建态/初始态/新建态(Created State/New State):创建一个进程一般包含创建 PCB 对象、初始化 PCB 必要的信息以及插入就绪队列 3 个步骤。当一个新进程被创建时,系统为其创建或分配 PCB,并初始化进程标识符等相关信息,但由于该进程所必需的某些资源尚未分配(如还未加载到内存、未为其分配必要的设备等资源),此时的进程已拥有了自己的 PCB,但进程自身还未进入主存,即创建工作尚未完成,进程还不能被调度运行,其所处的状态就是创建态。

引入创建态是为了保证进程的调度必须在创建工作完成后进行,以确保对进程控制块操作的完整性。同时,创建态的引入也增加了管理的灵活性,操作系统可以根据系统性能或主存容量的限制推迟创建态进程的提交。处于创建态的进程获得了其所必需的资源,并对其 PCB 初始化工作完成后,进程便可由创建态转入就绪态。

(5) 结束态/退出态/终止态(Exit State/Terminated State):进程的终止也要通过两个步骤。一是等待操作系统进行善后处理,包括释放内存、关闭文件和设备资源等;二是清除 PCB,并回收 PCB 资源返还给系统。当一个进程正常结束或异常中止后,无论是被操作系统还是其他有终止权的进程所终结,它都将进入终止态。进入终止态的进程不能再执行,但在操作系统中依然保留一个记录,保存了状态码和一些计时统计数据,供其他进程收集。一旦其他进程完成了对终止态进程的信息提取,操作系统将删除该进程。图 3.7 给出了增加了创建态和终止态后,进程的 5 种基本状态及其转换关系图。

(6) 挂起态。进程的挂起态是指进程被操作系统暂停执行,但进程全部或部分置换到外

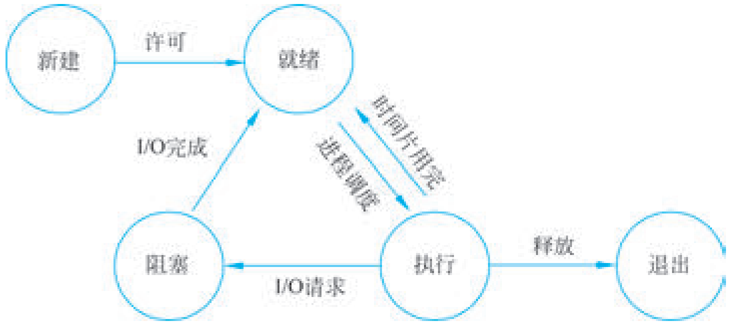

图 3.7 进程的 5 种基本状态及其转换

存中，以便在未来某个时间点恢复执行。挂起态用于满足如下需求。

① 负荷调节的需要。当实时系统中的工作负荷较重，可能影响对实时任务的控制时，可由系统把一些不重要的进程挂起，以保证系统能正常运行。

② 操作系统的需要。操作系统有时希望挂起某些进程，以便检查运行中的资源使用情况或进行记账。

③ 节能的需要。对于移动设备或服务器，挂起不必要的进程可以减少处理机和内存的功耗，延长电池寿命或降低散热负担。

④ 快速切换应用的需要。在桌面或移动操作系统中，挂起状态可以让用户快速切换应用程序，而无须每次都重新启动，提供了更加流畅的用户体验。

图 3.8 给出了具有挂起态的进程状态转换图。在引入挂起态后，进程需要增加从挂起态（又称为静止态）到非挂起态（又称为活动态）的双向转换。

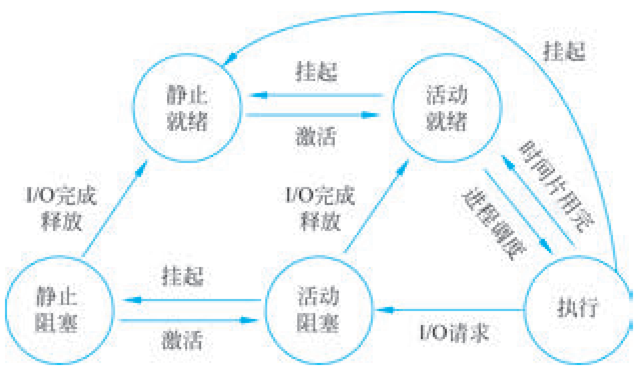

图 3.8 具有挂起态的进程状态转换图

- 活动就绪→静止就绪。当进程处于未被挂起的就绪态时，称为活动就绪态。当挂起该进程后，进程便转变为静止就绪态，处于静止就绪态的进程不再被调度执行。
- 活动阻塞→静止阻塞。当进程处于未被挂起的阻塞态时，称它处于活动阻塞态。当挂起后，进程便转变为静止阻塞态。处于该状态的进程在其所等待的事件出现后，将从静止阻塞态变为静止就绪态。
- 静止就绪→活动就绪。处于静止就绪态的进程，激活后该进程将转变为活动就绪态。
- 静止阻塞→活动阻塞。处于静止阻塞态的进程，激活后该进程将转变为活动阻塞态。
- 执行→活动就绪。正在运行的进程可以直接挂起到静止就绪态。
- 静止阻塞→静止就绪。处于静止阻塞状态的进程，当等待的 I/O 操作完成时，其可从静止阻塞态转换到静止就绪态。

3.4 进程控制

进程控制就是系统使用具有特定功能的程序段来对系统中的所有进程实施有效的管理，实现进程创建、撤销以及进程状态转换等功能。如果这些操作在运行过程中被打断，则可能会导致系统处于不一致的状态，例如部分资源被分配但未完全初始化，或者进程状态信息不正确。为此，需要使用原语(Primitive)完成相应操作。

原语是一种特殊的程序，它的执行具有原子性，一个原语操作中的所有动作，要么完全执行，要么完全不执行，该操作是不可分割的最小单位。本质上，原语操作必须一气呵成，不可中断，即不允许并发执行。在多线程或多进程环境中，原语操作可以保证数据的一致性和完整性，避免并发问题。进程控制相关操作均由原语完成，主要包括创建原语、撤销原语、阻塞原语、唤醒原语、挂起原语、激活原语等。

3.4.1 进程创建/创建原语

进程创建是操作系统中的一个关键操作，它涉及为一个新的程序实例分配资源并使其能够执行。进程可以由系统统一创建。例如，在批处理系统中，由操作系统的作业调度程序为用户作业创建相应的进程以完成用户作业所要求的功能。也可以由父进程创建，例如，在层次结构的系统中，父进程创建子进程以完成并行工作。系统统一创建的进程之间关系平等，一般不存在资源继承；而由父进程创建的进程之间则存在隶属派生关系。无论是用哪一种方式创建进程，在系统生成时，都必须由操作系统创建一部分承担系统资源分配和管理工作的系统进程。

无论是系统创建方式还是父进程创建方式，都必须调用创建原语来实现。创建原语扫描内存中存放 PCB 的链表，并申请新 PCB 空间，在申请到新 PCB 空间后，填入调用者提供的有关参数，最后形成代表新进程的 PCB 结构，其简单流程为：

(1) 查找 PCB 链表，查询是否有空闲 PCB 资源；

(2) 若没有，则创建失败；若有，则读取该空闲 PCB 的表项；

(3) 将相关参数填入 PCB 对应表项；

(4) 将当前 PCB 挂起到就绪队列中，同时将当前进程 PCB 加入进程家族树或进程链。

3.4.2 进程销毁/撤销原语

以下 3 种情况将导致进程被撤销：

(1) 该进程已完成所要求的功能而正常终止；

(2) 由于某种错误而导致非正常终止；

(3) 父进程要求撤销某个子进程。

无论因哪一种情况而导致进程被撤销，进程都必须释放它所占用的各种资源和 PCB 结构本身。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。另外，当一个父进程撤销某个子进程时，还需要审查该子进程是否还有自己的子孙进程，若有，则还需要撤销其子孙进程的 PCB 结构并释放它们所占有的资源。

在这一过程中，撤销原语首先检查 PCB 进程链或进程家族，寻找所要撤销的进程是否存在。如果找到了所要撤销的进程的 PCB 结构，则撤销原语在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。如果被撤销的进程有

自己的子进程，则撤销原语先撤销其子进程的 PCB 结构，并在释放子进程所占用的资源之后，再撤销当前进程的 PCB 结构和释放其资源。

3.4.3 阻塞与唤醒

进程的创建和撤销原语完成了进程从无到有和从存在到消亡的过程。而进程的阻塞原语和唤醒原语则分别表示将进程从运行态转换到阻塞态，从阻塞态转换到就绪态。

阻塞原语在一个进程等待某一事件(例如键盘输入数据、写盘、其他进程发来数据等)发生。当某事件尚不具备时，进程调用阻塞原语来阻塞自己。阻塞原语在阻塞一个进程时，由于该进程正处于执行状态，故应先中断处理机和保存该进程的处理机现场，然后将被阻塞进程置“阻塞”状态后插入等待队列，再转到进程调度程序选择新的就绪进程投入运行。阻塞原语的主要步骤为：保存当前进程的处理机现场；修改该进程 PCB 中的状态为阻塞态；将其挂到对应的事件阻塞队列中；执行进程调度程序。在这里，转到进程调度程序调度其他进程执行是很重要的，否则处理机将会出现空转而浪费资源。

唤醒原语的主要步骤为：当某个事件发生时，该事件对应的阻塞队列中所有的进程状态全部设置为就绪态；并将这些进程挂到就绪队列中；最后执行进程调度程序。当等待队列中的进程所等待的事件发生时，等待该事件的所有进程都将被唤醒。显然，一个处于阻塞态的进程不可能自己唤醒自己。

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程唤醒。当由事件发生进程唤醒时，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，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。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。

此外，当一个事件发生时，操作系统必须扫描整个阻塞队列，搜索等待该事件的进程。但在大型操作系统中，队列中可能有几百甚至几千个进程，遍历更新耗时较长。为此，实际系统会为每种事件对应一个阻塞队列。当事件发生时，相应队列中的所有进程都转换到就绪态。

3.4.4 挂起与激活

当出现了引起进程挂起的事件时，例如用户进程请求将自己挂起，或父进程请求将自己的某个子进程挂起，系统将利用挂起原语将指定进程或处于阻塞态的进程挂起。挂起原语检查被挂起进程的状态，若处于活动就绪态，则将其改为静止就绪；对于活动阻塞态的进程，则改为静止阻塞态；若被挂起的进程正在执行，则改为静止就绪态，并转向调度程序重新调度。

当发生激活进程的事件时，例如父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间，则可将在外存上处于静止就绪态的该进程换入内存。这时，系统将利用激活原语将指定进程激活。激活原语先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，则将其改为活动就绪态；若为静止阻塞态，便将之改为活动阻塞态。

3.4.5 Linux 下的进程控制系统调用

在 Linux 操作系统中，进程控制原语是操作系统内核提供给用户程序的接口，使得用户空

间的程序能够与操作系统交互，从而实现进程管理功能。这些系统调用主要包括创建新进程、执行新程序、等待子进程结束、终止进程、获取进程信息和改变进程属性。

（1）创建新进程：fork()创建一个新的子进程。新进程几乎完全复制了父进程的状态，包括代码段、数据段和堆栈等，但拥有独立的进程 ID。父子进程之间的区别在于返回值：fork()在父进程中返回子进程的 PID，在子进程中返回 0。vfork()类似于 fork()，但它更高效，因为它不会复制父进程的地址空间，而是让子进程暂时借用父进程的地址空间，这适用于子进程很快会调用 exec 的情况。clone()提供了比 fork()更细粒度的控制，可以指定哪些资源应该共享或分离。它是创建线程的基础，也用于一些高级进程创建场景。

以 fork()为例，当一个进程调用了 fork()时，操作系统会为新的子进程分配资源，包括但不限于内存空间、文件描述符等，并将父进程的数据段、堆栈段和寄存器状态复制给子进程。父进程和子进程各自拥有唯一的 PID。子进程的 PID 是正整数，而父进程中的 fork()调用则返回这个 PID。对于子进程来说，fork()返回 0。由于父子进程相对独立，因此它们可以分别并发执行代码。通常情况下，程序会在 fork()之后根据返回值的不同来区分是在父进程中还是子进程中，然后采取相应的行动。例如下面的代码：

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    // 创建子进程
    pid = fork();
    if (pid < 0) {              //错误处理
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {      //子进程
        printf("This is the child process. My PID is %d\n", getpid());
    } else {                    //父进程
        printf("This is the parent process. My PID is %d and my child's PID is %d\n", getpid(), pid);
    }
    return 0;
}
```

在这个例子中，如果 fork()调用失败，则它会返回一个负数，表示错误发生。如果 fork()成功并且返回的是 0，那么这段代码是在子进程中执行的，因为只有子进程从 fork()得到的结果是 0。如果 fork()返回的是正值，那么这段代码是在父进程中执行的，返回的正值是新创建的子进程的 PID。虽然父子进程开始时看起来是相同的，但它们实际上是独立的实体。任何对变量或数据结构的更改均不会影响对方，除非这些改变发生在共享资源上，例如共享内存或者文件描述符。当子进程结束时，如果父进程没有及时回收它的退出状态（通过 wait()或 waitpid()），那么子进程就会变成僵尸进程（zombie process）。僵尸进程仍然存在于进程表中，直到父进程回收其信息为止。如果父进程在子进程之前结束，那么子进程将成为孤儿进程（orphan process）。孤儿进程会被初始化进程（通常是 PID 为 1 的 init 进程）收养，并由它负责清理。

（2）执行新程序：exec*()家族是一组函数（如 execl()、execv()、execle()、execve()等），它们用一个新程序替换当前进程的映像。这意味着调用这些函数后，当前进程将不再执行原

来的代码，而是开始执行新加载的程序。例如，execv()用于在当前进程中加载并执行一个新的程序，替换当前进程的映像。这意味着调用 execv()后，当前进程将停止执行原来的代码，转而开始执行新加载的程序。execv()的特别之处在于它接收一个路径名和一个参数列表作为输入。

execv()不会自动继承当前进程的环境变量。如果需要传递环境变量给新程序，则应该考虑使用 execve()，它可以显式地指定环境变量。由于 execv()会替换当前进程的映像，因此在调用之前不需要担心清理资源的问题。但是，应该确保在此之前已经正确关闭不再需要的文件描述符等资源。当使用 execv()或相关函数时，请务必对输入进行适当的验证，以防止潜在的安全风险，如路径遍历攻击或命令注入。

(3) 等待子进程结束：wait()和 waitpid()用于等待一个或多个子进程结束，并回收其资源。wait()会阻塞直到任意一个子进程终止，而 waitpid()提供了更多的控制选项，例如可以选择等待特定的子进程或者非阻塞地检查是否有子进程已退出。以 waitpid()为例，其用于等待一个或多个子进程改变状态。它允许父进程暂停执行，直到指定的子进程终止或停止，并且可以获取该子进程的状态信息，这有助于避免僵尸进程，即那些已经结束但其退出状态尚未被父进程回收的子进程。如果不调用 waitpid()或类似的函数来回收子进程的状态，那么子进程在结束之后就会变成僵尸进程。僵尸进程仍然占用进程表中的条目，直到它们的状态被回收为止。如果父进程在子进程之前结束，那么子进程将成为孤儿进程，并由初始化进程(通常是 PID 为 1 的 init 进程)收养。如果不想让父进程一直阻塞等待子进程结束，则可以在调用 waitpid()时加上 WNOHANG 选项。这样，如果子进程还没有结束，则 waitpid()将立即返回 0，而不会阻塞。当有多个子进程时，可以通过传递不同的 PID 参数给 waitpid()来选择性地等待特定的子进程，或者使用−1 来等待任意一个子进程。

(4) 终止进程：exit()或_exit()用来正常终止进程。exit()会先清理环境变量和缓冲区再退出，而_exit()则直接退出，不进行任何清理工作。kill()和 raise()可以发送信号给其他进程或自身来改变进程的行为，例如终止进程。kill()发送信号到指定的进程，而 raise()则是向调用者自身发送信号。以 exit()为例，其用于终止当前进程，并返回一个状态码给父进程。这个状态码通常用来指示程序是否成功执行以及可能的错误原因。exit()会先调用所有已注册的 atexit()处理函数，然后刷新所有标准 I/O 流并关闭它们。接着，它会关闭所有打开的文件描述符。完成上述清理后，exit()调用_exit()来终止进程。这一步不会返回，因为进程已经结束。在某些情况下，exit()可能会发送 SIGCHLD 信号给父进程，通知它子进程的状态变化。在多线程程序中调用 exit()会导致整个进程立即终止，而不仅仅是终止调用它的那个线程。如果只想终止单个线程，则应使用 pthread_exit()。默认情况下，当进程接收到某些信号(如 SIGINT 或 SIGTERM)时，它会自动调用 exit()。

(5) 获取进程信息：getpid()返回调用者的进程 ID，getppid()返回调用者的父进程 ID。getuid()、geteuid()、getgid()、getegid()分别获取用户的实际 UID、有效 UID、GID 和有效 GID。uname()获取关于操作系统的名称和其他相关信息。

(6) 改变进程属性：setuid()、setgid()、setreuid()、setregid()分别用于更改进程的用户 ID 或组 ID。chdir()和 fchdir()用于改变当前工作目录。nice()和 setpriority()可以调整进程的优先级。

这些原语构成了 Linux 进程管理的基础，为开发者提供了丰富的工具来控制和管理进程生命周期中的各方面。每个系统调用都有其独特的使用场景和局限性，理解它们可以帮助我

们更好地编写稳定且高效的多任务程序。

读者可能会好奇,系统中第一个进程没有父进程,它又是如何创建的?Linux 完成自己的内核初始化工作后,控制权将完全转交给 init 进程;init 进程负责启动和管理系统中的其他进程,并且在系统启动时首先被创建。根结点 init 是所有进程的祖先,其进程 ID(PID)始终为 1。

不同于其他进程结点,Linux 在内核初始化工作完成后,会调用 init 函数(这个函数通常位于内核源码的 init/main.c 文件中),并通过 kernel_thread 或 do_execve 等机制创建和启动用户空间中的第一个进程——init 进程(内核会根据配置选择来源于 sbin、etc、bin 或/lib/systemd/system 等路径下的 init 程序,如果都不存在,则选择/bin/sh 启动)。内核将处理机的特权级别从内核态切换到用户态,使 init 进程能够在用户空间运行。init 进程的 PID 始终为 1,并根据配置文件(如 systemd 的/etc/systemd/system/default.target)启动系统中的各种服务和守护进程,确保系统能够正常运行,并且在整个系统生命周期中一直存在,直到系统关机或重启。init 进程肩负如下任务:①进程管理:init 进程负责启动和管理系统中的其他进程,它根据配置文件启动各种服务和守护进程。②孤儿进程的回收:当某个子进程的父进程终止时,该子进程会成为孤儿进程,init 进程会接管这些孤儿进程,并在其退出时回收其资源。③信号处理:init 进程可以接收和处理特定信号(如 SIGCHLD),以便及时回收僵尸进程。

跟踪状态(TASK_TRACED)

在 Linux 操作系统中,当一个进程被另一个进程(如调试器)通过 ptrace()系统调用进行跟踪时,这个被跟踪的进程就会进入 TASK_TRACED 状态。在这种状态下,进程暂停了正常的执行流程,并等待跟踪它的进程对它进行操作。例如,在使用 GDB 这样的调试工具时,当设置了一个断点并且被调试的程序运行到该断点时,被调试的程序将进入 TASK_TRACED 状态。处于 TASK_TRACED 状态的进程不会继续执行任何代码,只有当跟踪进程(如调试器)发出特定的 ptrace 命令(如 PTRACE_CONT 或 PTRACE_DETACH),或者跟踪进程退出后,被跟踪的进程才能恢复正常执行。跟踪进程可以利用 ptrace 提供的接口来检查和修改被跟踪进程的内存、寄存器内容以及控制其执行流程(如单步执行、继续执行等)。在实际开发中,TASK_TRACED 状态主要用于支持调试功能。开发者可以利用这一机制深入理解程序的行为,定位并修复错误。此外,它也是实现一些高级特性(如性能分析、安全监控等)的基础。

TASK_TRACED 状态和挂起状态在 Linux 操作系统中代表进程的两种不同情况,尽管它们都涉及进程的暂停执行,但各自的目的和机制有所不同。TASK_TRACED 是一种特殊的进程状态,当一个进程正在被另一个进程跟踪时(通过调试器,如 GDB),它会进入这种状态。在这种状态下,目标进程暂停执行,并等待跟踪它的进程进行进一步操作。这通常发生在调试过程中,例如设置断点后,程序运行到该断点处就会进入 TASK_TRACED 状态。挂起状态是另一种进程状态,它指的是进程暂时被淘汰出内存的状态。这意味着进程的数据从物理内存移到了外存(如硬盘上的交换区)。挂起可以由用户请求、父进程请求、操作系统的需要、负载调节的需要等原因引起。

尽管两者的主要用途不同,但在某些情况下,一个进程可能会同时经历这两种状态。例如,在调试过程中,如果系统面临内存压力,那么即使是处于 TASK_TRACED 状态的进程也可能被操作系统暂时挂起,以释放宝贵的内存资源。然而,这种情况相对少见,因为大多数调试工具和操作系统都会尽量避免对调试中的进程施加额外的压力。

3.5 线程

进程作为操作系统中资源分配和调度的基本单位，尽管为多任务处理提供了基础支持，但在实际应用中也暴露出了一些不足之处。首先，进程资源开销大。每个进程都有自己独立的地址空间、内存、数据栈以及其他用于跟踪执行的辅助数据，这意味着创建、销毁和切换进程时都需要较大的系统开销。其次，进程之间的通信相对复杂，因为它们各自拥有独立的内存空间。要实现进程间的通信(IPC)，通常需要额外的机制，如管道、共享内存、消息队列等。这不仅增加了编程的复杂性，而且在跨进程传递大量数据时效率较低，相关内容将在第4章进行介绍。再次，某些情况下进程响应时间较长，当一个进程中的某个任务发生阻塞(例如等待I/O操作完成)时，整个进程可能会被挂起，导致即使进程中其他不依赖于该资源的任务也无法继续执行。这种情况下，系统的响应时间和吞吐量都会受到影响。最后，进程的并发度仍不够高。在单处理器或多核处理器环境下，仅靠进程来实现并发执行并不能充分利用硬件资源。由于进程间切换的成本较高，因此频繁地切换进程会导致处理机的利用率下降。

3.5.1 线程的引入与概念

为解决上述问题，并进一步提升并发处理能力，科学家提出了线程(Thread)的概念。线程可以被视为进程内部更轻量级的执行单元。与进程不同，线程共享相同的内存空间和资源，但它们独立执行，并且每个线程都有自己的执行上下文(如程序计数器、寄存器和栈)。这使得线程之间的切换比进程更为高效。例如，在文件服务进程中设置了多个服务线程，当第一个线程等待时，第二个线程可以继续运行以提供服务。

“线程”一词大约是在1967年首次提出，但直到20世纪80年代中期，随着UNIX和其他操作系统开始正式支持线程，线程才逐渐成为操作系统和编程语言中的一等公民。例如，1988年的SunOS 4.0版本引入了轻量级进程(LWP)，这是后来POSIX线程标准的基础之一；而Windows NT自其发布以来就支持多线程编程。

引入线程后，线程是操作系统能够进行运算调度的最小单位，它比进程更轻量级，可以提高程序的并发性和响应速度。在实现细节上，线程和进程有着显著的区别，尤其是在堆栈、寄存器、局部变量等方面，如图3.9所示，每个进程都有独立的堆栈空间，用于存储函数调用时的

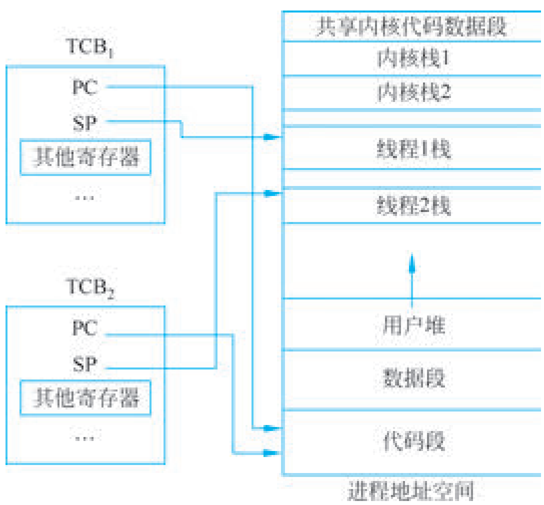

图3.9 多线程进程内存映像结构

返回地址、参数以及局部变量等信息。而同一进程内的所有线程共享该进程的地址空间，但每个线程都有自己独立的堆栈。这意味着虽然线程可以访问相同的全局数据，但它们各自的局部变量、函数参数及返回地址都是分开存放的，从而保证了线程间的隔离性。例如，在 Java 虚拟机中，每个线程都有自己的一份虚拟机栈和本地方法栈，而堆和方法区则是由所有线程共同使用的。

同一个进程中定义的局部变量通常位于该进程的私有堆栈段内。由于进程之间相互隔离，因此一个进程中的局部变量不会影响其他进程。而对于同一进程内的不同线程而言，它们的局部变量也是各自独立的。即使是在同一个函数内部声明的局部变量，不同的线程也会拥有自己的副本。这种设计确保了即使是在多线程环境中，局部变量也能够保持线程安全，即一个线程的操作不会意外地干扰到另一个线程的数据。

线程的主要特征如下。

(1) 线程是一个轻型实体，它不拥有系统资源，但每个线程都应有一个唯一的标识符和一个线程控制块，线程控制块记录了线程执行的寄存器和栈等现场状态。

(2) 不同的线程可以执行相同的程序，即当同一个服务程序被不同的用户调用时，操作系统会为它们创建不同的线程。

(3) 同一进程中的各个线程共享该进程所拥有的资源。

(4) 线程是处理器的独立调度单位，多个线程是可以并发执行的。在单处理机的计算机系统中，各线程可交替地占用处理机；在多处理机的计算机系统中，各线程可同时占用不同的处理机，若各个处理机同时为一个进程内的各线程服务，则可缩短进程的处理时间。

3.5.2 线程和进程的区别

进程和线程是操作系统中的两个核心概念，它们都是为了实现程序的并发执行而设计的。尽管两者之间存在一定的联系，但由于实现上的巨大差异，它们在多方面有着本质的区别。

(1) 在内存分配与共享方面，进程之间具有严格的边界划分，每个进程都有自己独立的地址空间，其中包括代码段、数据段、堆区和栈区。为实现进程间的通信，必须通过专门的机制，如管道、消息队列、信号量或套接字等。而在同一进程内的线程共享几乎所有的系统资源，包括文件描述符、信号处理器、打开的文件列表等。只有那些明确属于线程自身的属性(如上述提到的寄存器、堆栈)才是线程特有的。

(2) 在进行上下文切换时，进程之间的切换涉及完整的上下文环境变化，包括处理机寄存器的内容，因此每次进程切换都需要保存当前进程的所有寄存器状态，并加载下一个要运行进程的相应值，这导致了较大的开销。相比之下，线程切换只需要更改少量特定于线程的状态信息，如程序计数器、栈指针以及其他少量寄存器。这是因为多个线程共享同一个进程的大部分资源，减少了需要保存和恢复的数据量，使得线程间的切换更加高效。

(3) 在调度方面，进程在传统的操作系统中既是资源的拥有者，也是系统调度和分配的单位。而在引入线程之后，进程仅作为资源的分配单位，线程作为调度和分配的单位。在同一进程内，线程的切换不会引起进程的切换。而由一个进程中的线程切换到另一个进程中的线程时，才会发生进程切换。

(4) 在并发性能方面，在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程内的多个线程之间亦同样可以并发执行，因此使系统具有了更好的并发性，系统资源利用率和系统吞吐量得到了提高。

(5) 在系统资源分配方面，在引入线程的操作系统中，线程成为被调度和分配的基本单位。线程基本不拥有资源，只有一些运行所必需的资源，但它可以和进程内的其他线程共享进程所拥有的全部资源，即拥有一个进程的代码段、数据段以及系统资源(如打开的文件、I/O 设备等)。

(6) 在系统开销方面。在创建和撤销进程时，系统都要进行资源分配和回收，如内存空间、I/O 设备等。因此，操作系统所付出的时空开销显著地大于线程创建和撤销的开销。类似地，在进程进行切换时，涉及当前进程的现场保存，以及新选择进程的上下文恢复。而线程的切换只需要保存和设置少量的寄存器内容，并不涉及存储器管理方面的操作。由此可见，进程间切换的开销也大于线程间切换的开销。此外，由于同一进程内的多个线程具有相同的地址空间，因此线程之间的同步和通信也变得更加容易。

3.5.3　线程的实现方式

线程的实现方式根据不同的操作系统和编程语言而有所不同，分为用户级线程、内核级线程和混合模式。其主要区别是在用户态还是系统态来实现线程的管理。混合模式结合了上述两种方法的优点，既存在用户级线程，也存在内核级线程。在这种模式下，多个用户级线程可以被映射到少数几个内核级线程，形成多对多的关系。这样既能享受用户级线程带来的低开销优势，又能利用内核级线程提供的高效调度能力。此外，它还可以更好地支持多处理器系统中的并行计算。

用户级线程(user-level threads)的管理过程全部由用户程序完成，操作系统内核只对进程进行管理。由于这些线程都设置在用户空间，而线程所执行的操作也无须内核的帮助，因此内核完全不知道用户级线程的存在。为对用户级线程进行管理，操作系统提供了一个在用户空间执行的线程库。该线程库提供创建、调度、撤销线程功能。同时，该线程库也提供线程间的通信、线程的执行以及存储线程上下文的功能。当一个线程被创建时，线程库为其生成相应的线程控制块等数据结构，为线程控制块中的参数赋值，并把该线程置于就绪态，其处理过程与进程创建过程大致相似，不同之处如下。

(1) 用户级线程的调度算法和调度过程全部由用户自行选择和确定，与操作系统内核无关。在用户级线程系统中，操作系统内核的调度单位仍是进程。如果进程的调度区间为 T，则在 T 区间内，用户可以根据自己的需要设置不同的线程调度算法。

(2) 用户级线程的调度算法只进行线程上下文切换，而不进行处理机切换，且其线程上下文切换是在内核不参与的情况下进行的。也就是说，线程上下文切换只是在用户栈、用户寄存器等之间进行切换，不涉及处理机的状态。

(3) 因为用户级线程的上下文切换与内核无关，所以可能出现如下情况：即当一个进程因 I/O 中断或时间片用完等导致该进程退出处理机时，属于该进程的执行中线程仍处于执行态。也就是说，尽管相关进程的状态是阻塞的或等待的，但所属线程状态却是执行的。

内核级线程(kernel-level threads)由操作系统内核进行管理。操作系统内核给应用程序提供相应的系统调用和应用程序接口 API，以使用户程序可以创建、执行、撤销线程。

与用户级线程不同，内核级线程既可以被调度到一个处理机上并发执行，也可以被调度到不同的处理机上并行执行。操作系统内核既负责进程的调度，也负责进程内不同线程的调度工作。因此，内核级线程不会出现进程处于阻塞态而线程处于执行态的情况。

另外，内核级线程技术也可用于内核程序自身，从而提高操作系统内核程序的执行效率。

在同一个进程中，从一个线程切换到另一个线程时，处理机需要从用户态转换到内核态，这是因为用户进程的线程在用户态运行，而线程调度和管理是在内核态实现的。因此，内核级线程的上下文切换时间要大于用户级线程的上下文切换时间。

3.5.4 POSIX 线程库实现

下面以 pthread 库为例，简单介绍线程的具体实现。在 C 语言中使用 pthread 库时，线程的实现既可以是内核级线程，也可以是用户级线程，这取决于操作系统的版本和支持情况。

自 Linux 2.6 内核引入本地 POSIX 线程库(Native POSIX Thread Library, NPTL)以来，它成为 Linux 系统中线程实现的标准方式。NPTL 采用了一对一的映射模型，即每个用户空间中的线程都对应着一个独立的内核调度实体。这意味着每当调用 pthread_create 创建一个新的线程时，内核就会相应地创建一个可调度的线程，并且这些线程可以直接由操作系统内核进行管理。这种设计使得线程能够充分利用多处理器或多核心处理器的优势，同时提供了良好的性能和可靠性。此外，由于线程与内核之间的直接关联，因此如果某个线程被阻塞(如等待 I/O 操作完成)，则其他线程仍然可以继续执行，不会影响整个进程的运行。

为实现这一点，NPTL 内部实际上是利用了 Linux 的 clone()系统调用来创建轻量级进程(LWP)。clone()允许开发者指定多个选项来控制新进程中的哪些资源应该共享或复制，从而实现了类似线程的行为。例如，所有线程共享同一个地址空间、文件描述符表等，但每个线程都有自己独立的栈和其他必要的上下文信息。

尽管现代 Linux 系统普遍支持内核级线程，但在早期版本(如 Linux 2.4 及以前)中，线程是通过用户级线程库实现的。这类线程完全由应用程序库负责创建、调度和销毁，而不需要任何来自操作系统的帮助。因此，它们具有较低的创建成本和更快的上下文切换速度。然而，这种方式也带来了局限性：因为内核并不知道用户级线程的存在，所以一旦任何一个线程发生阻塞，整个进程都会受到影响，无法有效地利用多核处理器的能力。

在实践中，即使是在支持内核级线程的操作系统上，某些特殊的场景下也可能需要使用用户级线程。例如，在嵌入式操作系统或其他资源受限的环境中，可能更倾向于选择轻量级的用户级线程以减少内存占用和提高响应速度。不过，随着硬件性能的提升和技术的发展，这类需求已经变得越来越少。

无论底层是内核级还是用户级线程，pthread 库都为开发者提供了一套统一的 API 接口，用于创建、管理和同步线程。这些 API 不仅符合 POSIX 标准，确保了跨平台兼容性，而且抽象掉了具体的实现细节，使得开发人员不必关心线程是由内核还是用户态库管理的。例如，pthread_create()用于创建新线程，pthread_join()用于等待线程结束，而 pthread_mutex_*()系列函数则提供了互斥锁机制，保证了线程间的正确协作。

综上所述，在当前大多数 Linux 发行版中，通过 pthread 库创建的线程都是基于 NPTL 实现的内核级线程，除非特别指定了不同的线程库或者运行环境不支持内核级线程。可以通过检查系统配置来确认是否启用了 NPTL，例如执行命令 getconf GNU_LIBPTHREAD_VERSION 可以查看当前使用的线程库版本。此外，值得注意的是，虽然理论上存在用户级线程的可能性，但在实际应用中，尤其是在服务器端编程领域，几乎所有的多线程程序都是以内核级线程的形式运行的。

Python 的全局解释器锁

Python 的全局解释器锁(Global Interpreter Lock, GIL)是 CPython 解释器的一个机制，

它可以确保任何时候只有一个线程在执行 Python 字节码。这意味着即使在多核处理器上运行,Python 的多线程程序也无法实现真正的并行执行,特别是在处理 CPU 密集型任务时。因此,在这种上下文中,Python 的多线程有时被称为“伪多线程”。

GIL 最初是由 Python 的创始人 Guido van Rossum 引入的,主要是为了简化内存管理。由于 Python 使用引用计数进行垃圾回收,因此要求对对象的引用计数进行原语操作以避免竞争条件。GIL 提供了一个全局锁,使得任何时候只有一个线程能够执行 Python 代码,从而解决了这个问题。

对于 I/O 密集型的任务,如网络请求或文件读写,GIL 的影响较小,因为当一个线程等待 I/O 操作完成时,它会释放 GIL,允许其他线程继续执行。然而,对于 CPU 密集型任务,例如复杂的数学计算或者数据处理,GIL 成为性能瓶颈,因为这些任务无法利用多核处理器的优势来加速执行。

为避免 GIL 带来的性能问题,开发者可以使用 Python 的标准库 multiprocessing 模块,其允许创建多个独立的进程,每个进程都有自己的解释器实例和内存空间,因此不受 GIL 的影响。这种方法适合于 CPU 密集型任务。另一种方式是编写 C 扩展模块,在 C 代码中运行多线程程序,这样就可以在 C 代码中执行并行计算,然后返回给 Python 环境。此外,concurrent. futures 也提供了一个高层次的接口来异步执行函数调用,它包含两个类:ThreadPoolExecutor 和 ProcessPoolExecutor。前者使用线程池,后者使用进程池,并且不受 GIL 的影响。

多线程编程小技巧

(1) 数据局部性优化:数据局部性优化是提高程序性能的关键策略之一,尤其是在现代计算机体系结构中,处理机访问内存的速度远低于执行指令的速度。为最大化利用处理机缓存,减少缓存未命中(cache miss)的次数,开发者需要精心设计数据结构和算法以增强数据的局部性。例如,可以优先选用连续存储的数据结构,如数组、非链表等非连续存储结构,以更好地利用空间局部性;通过分割大循环为较小的子循环来改善数据局部性,使数据加载到缓存后尽可能多地被使用;改变嵌套循环的顺序,以便更好地利用空间局部性;选择或设计能够有效利用缓存特性的算法,如快速排序比归并排序更有利于缓存使用。

(2) 避免伪共享:伪共享(false sharing)是并发编程中的一个隐蔽但严重的性能问题,它发生在多线程程序中,当不同的线程修改位于同一缓存行的不同变量时,尽管这些变量在逻辑上是独立的,但由于它们共享同一个缓存行,导致频繁的缓存同步操作,从而影响了程序性能。为避免伪共享,可以通过在数据结构中添加额外的填充字段来确保每个需要保护的变量独占一个缓存行;将经常一起使用的变量放在同一个缓存行中,而将不相关的变量分开到不同的缓存行;设计和使用无锁的数据结构可以帮助减少线程间的竞争,降低伪共享的风险。

(3) 使用 SIMD 指令集:SIMD(Single Instruction Multiple Data,单指令多数据流)指令集是一种并行计算技术,它允许一个单独的指令对多个数据点执行相同的操作。这种技术特别适用于多媒体处理、科学计算、数据分析等领域,其中大量相似的数据需要进行相同的运算。例如,在图像处理中,一个 SIMD 加法指令可以同时将 4 个像素值相加,而不是依次处理每个像素。在高性能计算领域,常常会结合使用 SIMD 和多线程技术来最大化性能。例如,在一个多线程环境中,每个线程都可以利用 SIMD 指令集来加速其内部的数据处理;设计时应尽量减少共享状态,或者采用无锁的数据结构和算法,这样不仅可以减少锁竞争带来的性能损耗,还能让 SIMD 指令更高效地发挥作用。

关键术语

阻塞态	特权态	挂起态	子进程
父进程	交换	退出态	运行态
进程	用户模式	就绪态	抢占
新建态	中断	线程	多线程
进程映像	进程上下文切换	轻量级进程	内核级线程
进程控制块	系统态	内核模式	模式切换
程序状态字	消息	用户级线程	

习题

1. 结合具体场景谈谈线程概念的动机及优势。
2. 阐述进程的模式切换和上下文切换的区别和联系。
3. 什么是进程？操作系统为何要引入进程？调研 Linux 等系统，并结合实际详细阐述进程的基本要素及其在内存中的映像布局。
4. 进程最基本的状态有哪些？哪些事件可能引起不同状态之间的相互转换？
5. 为何要引入进程的挂起态？
6. 结合 PCB 知识，描述当进程发生如下状态转换时，操作系统要修改 PCB 中的哪些内容？

就绪→运行　　运行→阻塞

7. 在创建一个进程时，操作系统完成的主要工作是什么？哪些事件会引起进程的创建？
8. 在终止一个进程时，操作系统完成的主要工作是什么？哪些事件会引起进程的终止？