第二章
进程、线程与作业

2.1 多道程序设计

多道程序设计（multi-programming）是操作系统所采用的最基本、最重要的技术，其根本目标是提高整个计算机系统的效率。衡量系统效率有一个尺度，这就是吞吐量。系统的吞吐量定义为单位时间内系统所处理的作业（程序）的道数（数量），即

$$\text{吞吐量}=\frac{\text{作业道数}}{\text{全部处理时间}}$$

不难理解，系统的效率与系统中处理器等资源的利用率有着密切的关系。如果系统的资源利用率高，则单位时间内所完成的有效工作多，吞吐量大；反之，如果系统的资源利用率低，则单位时间内所完成的有效工作少，吞吐量小。所以，提高系统的吞吐量应当从提高系统的资源利用率入手。


多道程序设计

下面说明多道程序设计将在很大程度上提高系统的资源利用率，从而提高吞吐量，即效率。

2.1.1 单道程序设计的缺点

所谓单道程序设计，就是一次只允许一个程序进入系统的程序设计方法。当然，单道程序设计是简单的，但是它有一个严重的缺点：资源利用率极低。具体表现在以下几个方面。

1．设备资源利用率低

现代计算机系统都配备许多外围设备，如扫描仪、绘图仪、打印机、磁带、磁盘等。容易理解，在单道程序系统中，内存中仅存在一个程序，该程序一般只能用到这些设备集合的一个子集，而很多未被用到的设备则会闲置。即使一个程序用到设备集合中的全部设备，这些设备通常也不会自始至终地处于忙碌状态。因此，那些未被用到的设备资源便会浪费，这种浪费在单道程序系统中是不可避免的。

2．内存资源利用率低

随着硬件技术的提高和成本的下降，内存容量在不断增加，目前已经达到几百兆字节或上

千兆字节，而一般程序的长度远远小于内存容量。在此情况下，若仍然采用单道程序设计，内存空间的浪费将是惊人的。

3．处理器资源利用率低

在单道程序系统中，如果运行程序不执行输入输出操作，则处理器的利用率一般不受影响。然而不与任何外围设备打交道的程序是很少的，绝大多数程序都需要通过输入输出操作与外部交往。另外，读写文件等操作也要与磁盘、光盘等设备打交道，所以应当考虑输入输出操作对CPU 利用率的影响。

中断、通道、DMA 控制器的引入使处理器与外围设备的并行成为可能。当需要执行输入输出操作时，处理器启动设备、通道或 DMA 控制器，然后便可以转做其他事情。当输入输出操作完成时，设备（通道或 DMA 控制器）发出中断请求，处理器接到中断请求信号后进行相应的处理，然后继续执行被中断的程序。如图 2-1 所示，在 t_2 时刻处理器启动设备，在 t_5 时刻设备发出完成中断的信号。原则上说，在 t_2～t_5 期间内，处理器可以做任何有意义的计算工作。假设在 t_5～t_6 期间内处理器所完成的工作可以提前，即它不需要等待数据传输完成，则可以将其安排在 t_2～t_5 期间内做，这样可以提高处理器的利用率。但是，假若在 t_5～t_6 期间内处理器所进行的工作需要等待数据传输的结果，则工作无法提前。也就是说，在 t_2～t_5 期间内处理器将无事可做。

图 2-1 单道程序设计中的处理器利用情况

应当指出，如果数据传输的速度很快，时刻 t_2 和 t_5 之间的间隔很短，即 $t_5-t_2 \ll t_2-t_1$，处理器资源的浪费是可以容忍的。但是，通常设备的传输速度都很慢，即 $t_5-t_2 \gg t_2-t_1$，处理器与外围设备的速度不匹配，由此而造成的处理器资源的浪费是惊人的。

单道程序的另一个缺点是不能表达程序内在的并行性。有些程序具有内在的并行性，用顺序程序模式很难刻画。例如，在 Windows 里的 Word 字处理程序，有如下 3 段代码需要并行执行：① 交互编辑，在缓冲区内交互式地编辑用户输入；② 语法检查，对缓冲区内用户输入的内容进行语法检查，若有错误则在下面画出一条红线；③ 定时保存，将缓冲区中的内容定时保存到磁盘上，以免故障时丢失信息。这 3 段程序如何安排它们的执行次序呢？无法安排，它们内在就是并行的。

2.1.2 多道程序设计的提出

造成上述资源利用率低下的根本原因是系统中的资源数量多，而资源使用者数量少。由此

人们想到，如果允许多个程序同时进入系统，即增加资源使用者的数量，则资源利用率应当能够得到提高，这就是多道程序系统思想的由来。简单地说，多道程序设计就是让多个程序同时进入系统并投入执行的一种程序设计方法。下面具体分析多道程序设计对于系统的资源利用率所带来的影响。

1．设备资源利用率提高

容易看出，如果允许若干个搭配合理的程序同时进入内存，这些程序分别使用不同的设备资源，则系统中的各种设备资源都会被用到并经常处于忙碌状态。设备资源利用率将得到明显提高。

2．内存资源利用率提高

显而易见，允许多道程序同时进入系统可以避免单道程序过短而内存空间过大所造成的存储空间的浪费，从而提高内存资源的利用率。同时，进入系统的多个程序可以保存在内存的不同区域中。

3．处理器资源利用率提高

如前所述，在单道程序系统中，处理器资源利用率低的主要原因是当运行程序等待输入输出操作完成时，处理器无事可做。由此自然想到，如果将两道程序同时放入内存，在一个程序等待输入输出操作完成期间，处理器执行另一个程序，这样便可提高处理器的利用率，例如，在图 2-1 所示的例子中，增加一道程序 *B*，该程序便可利用空闲的处理器时间，如图 2-2 所示。在时刻 t_2，程序 *A* 放弃处理器，但是程序 *B* 暂不具备运行条件，在时刻 t_3，程序 *B* 获得 CPU 并运行。在时刻 t_4，程序 *B* 启动输入输出设备 2，然后等待数据传输完成。

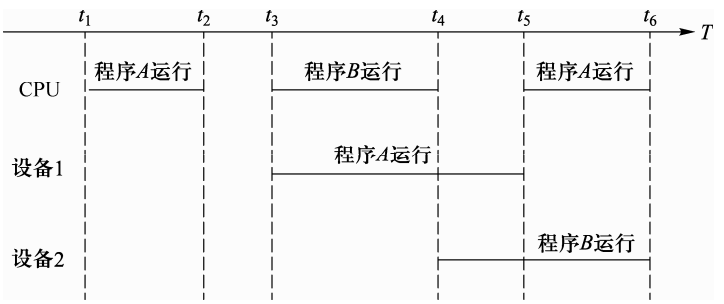

图 2-2 两道程序系统中的处理器利用情况

比较图 2-1 与图 2-2 不难发现，在单道程序系统中，处理器在 $t_2 \sim t_5$ 期间内完全被闲置，而在两道程序系统中，处理器在 $t_2 \sim t_5$ 期间内有一段时间，即 $t_3 \sim t_4$ 得到利用。也就是说，与单道程序相比，两道程序提高了处理器资源的利用率。

容易看出，如果增加内存中程序的道数，处理器资源的利用率可以进一步提高。从理论上说，当内存中程序的道数充分多时，处理器的利用率可以达到 100%。

根据上述分析可知，多道程序设计可以有效地提高系统中各种资源的利用率，从而提高系

统吞吐量。那么内存中的程序数量是否越多越好呢？答案是否定的。首先，内存的容量限制了系统可以同时处理程序的数量。其次，物理设备的数量也是一个制约条件，如果内存中同时运行的程序过多，这些程序可能会相互等待被其他程序占用的设备资源，反而会影响系统效率。另外，内存中过多的程序会形成对处理器资源的激烈竞争，既可能影响系统的响应速度，也会增加因处理器分配而带来的系统开销。一般来说，要确定内存中同时接纳的程序的数量，主要应当考虑计算机的配置情况。

4．自然地表达一个程序内在的并行性

对于前面所举的 Word 程序的例子，用并行的方式可以给出自然的解决方案，我们定义一个进程，其中包含 3 个线程，分别执行交互编辑、拼写检查、定时存盘这 3 个任务，三者之间没有执行次序，是内在并行的，不仅表达方便，而且实现高效。

2.1.3 多道程序设计的问题

如前所述，多道程序设计改善了系统资源的使用情况，从而增加了吞吐量，提高了系统效率，但是同时也带来了新的问题，即资源竞争。多道程序设计的实现必须协调好运行程序与资源之间的关系，在不发生混乱的前提下，使各种资源的使用效率尽量提高。具体来说，需要解决以下问题。

1．处理器资源管理问题

如果在任意时刻，系统中只有一个程序可以运行，问题比较容易解决，可以将处理器分配给这个唯一具备运行条件的程序。但是，由于可运行程序的数量一般远远多于处理器的数量，这就需要解决可运行程序与处理器资源竞争的问题，或者说，需要对处理器资源加以管理，实现处理器资源在各个程序之间的分配和调度。

2．内存资源管理问题

为使多个程序在一个系统中共存，需要按某种原则对内存空间进行划分，并将其分配给各个程序。由于一个程序在内存中的实际存放位置在其装入系统之前无法确定，而且在程序运行过程中地址还可能发生变化，因而程序只能使用相对地址，不能使用绝对地址。操作系统负责存储空间的分配与管理，并在硬件的支持下将程序产生的逻辑地址映射到内存空间的物理地址上，即实现程序的重定位。

此外，还应防止内存中多道程序之间的相互干扰或者对操作系统的干扰，即不允许一个程序侵犯另一个程序的地址空间或操作系统空间。为此，操作系统应当在硬件的支持下实现对存储空间的保护。

3．设备资源管理问题

用户都希望内存中的多道程序在使用设备时不发生冲突，即使它们在同一时刻使用系统中的不同设备资源，而对相同设备资源的使用在时间上应该尽量错开。尽管操作系统在选择程序进入系统时可以使进入系统的程序搭配得相对合理，但是由于程序使用资源的不确定性以及程序推进速度的不确定性，上述理想情况是很难达到的。也就是说，多个程序同时要求使用同一

资源的情况会经常发生，这就要求操作系统确定适当的分配策略，并据此对资源加以管理。

2.2 进程的引入


进程的引入

在多道程序系统中运行的程序是处于时断时续的状态之中的。一个程序获得处理器资源后向前推进，当它需要某种资源而未得到时只好暂停下来，以后得到所申请的资源时再继续向前推进。如此，一个程序的活动规律是：

推进 → 暂停 → 推进 → 暂停 → ……

当程序暂停时，需要将其现场信息作为断点保存起来，以便以后再次推进时能够恢复上次暂停时的现场信息并从断点处开始继续执行。这样，在多道程序系统中运行的程序需要一个保存断点现场信息的区域，而这个区域并不是程序的组成部分，因此就需要一个能够更准确地描述多道程序系统中执行程序的术语，这就是进程（process）。

2.2.1 进程的概念

进程的概念起源于 20 世纪 60 年代初期，首先由美国麻省理工学院的 Multics 系统和 IBM 公司的 CTSS/360 系统引入，得到人们的普遍重视，并被其后的操作系统研究者和设计者们广为采用。目前，进程已经成为操作系统乃至并发程序设计中一个非常重要的概念。不过，关于什么是进程，目前尚无统一的定义。确切地说，关于进程有许多解释，这些解释并不是完全等价的。

① 进程是程序的一次执行。

② 进程是可以参与并发执行的程序。

③ 进程是程序与数据一道通过处理器执行时所发生的活动。

④ 所谓进程，就是一个程序在给定的空间和初始环境下，在一个处理器上执行的过程。

1978 年，在庐山召开的国内操作系统研讨会上给出进程的定义如下。

定义 2-1 进程是具有一定独立功能的程序关于一个数据集合的一次运行活动。

上述关于进程的解释虽然各有所侧重，在本质上却是相近的，即都强调程序的执行，也就是进程的动态特性，这是进程与程序之间本质上的差异。尽管关于进程至今尚无公认和严格的定义，但是它已经被成功地用于操作系统构造以及并发程序设计中。读者应当在后面的讲述中仔细体会“进程”这一概念的含义。

2.2.2 进程状态及状态转换

1. 进程的状态

进程在其生存周期内可能处于以下 3 种基本状态之一。

① 运行（run）态：进程占有处理器资源，正在运行。显然，在单处理器系统中任一时刻只能有一个进程处于此种状态。

② 就绪（ready）态：进程本身具备运行条件，但是由于处理器的数量少于可运行进程的数量，暂未投入运行，即相当于等待处理器资源。

③ 等待（wait）态：也称为挂起（suspend）态、阻塞（block）态、睡眠（sleep）态。进程本身不具备运行条件，即使分给其处理器也不能运行。进程正在等待某一事件的发生，如等待某一资源被释放，等待与该进程相关的数据传输的完成信号等。

运行、就绪、等待是进程最基本的 3 种状态，对于一个具体的系统来说，为了实现某种设计目标，进程状态的数量可能多于 3 个。

2．状态转换

进程的 3 个基本状态之间是可以相互转换的。具体地说，当一个就绪进程获得处理器时，其状态由就绪态变为运行态；当一个运行进程被剥夺处理器资源时，如用完系统分给它的时间片，或者出现高优先级别的其他进程，其状态由运行态变为就绪态；当一个运行进程因某事件受阻时，如所申请资源被占用、启动数据传输未完成，其状态由运行态变为等待态；当所等待的事件发生时，如得到被申请资源、数据传输完成，其状态由等待态变为就绪态。进程间的基本状态转换关系如图 2-3 所示。进程状态转换是由操作系统完成的，对用户而言是透明的。一个进程在其生存周期内会经过多次状态转换，这体现了进程的动态性和并发性。

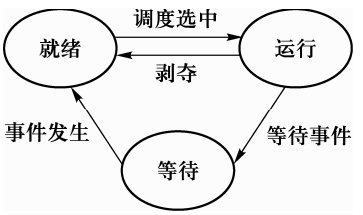

图 2-3 进程间的基本状态转换关系

对于不同的处理器调度算法，上述进程转换图可能略有不同。例如对于非抢先调度，不存在由运行态到就绪态的状态转换。


进程控制块

2.2.3 进程控制块

前面已经提到，多道程序系统中运行的程序需要有一个断点现场保存区域，这个区域就设在进程控制块（process control block，PCB）中。进程控制块是进程存在的标志，它由一组信息构成。除现场外，通常还包括许多其他内容，这些信息是系统对进程进行管理所需要的。

进程标识
用户标识
进程状态
调度参数
现场信息
家族联系
程序地址
当前打开文件
消息队列指针
资源使用情况
进程队列指针

图 2-4 进程控制块

定义 2-2 进程控制块是标志进程存在的数据结构，其中包含系统对进程进行管理所需要的全部信息。

对于不同的操作系统来说，进程控制块中信息的内容和数量不尽相同。一般来说，系统规模越大，功能越强，其进程控制块中信息的数量就越多。图 2-4 中列出了一般操作系统中进程控制块所应包含的项目。

在进程控制块中，“进程标识”通常为一个整数，称为进程号，用于区分不同的进程。“用户标识”通常也为一个整数，称为用户号，用于区分不同的用户。一个进程号与唯一的用户号相对应，而一个用户号可以与多个进程号相对应，即一个用户可以同时拥有多个进程。“进程状态”在就绪、运行、等待之间动态变化。“调度参数”

用于确定下一个运行的进程。“现场信息”用于保存进程暂停的断点信息，包括通用寄存器、地址映射寄存器、PSW、PC。“家族联系”记载本进程的父进程。“程序地址”记载进程所对应程序的存储位置和所占存储空间大小，具体内容与存储管理方式有关。“当前打开文件”用于记载进程正在使用的文件，通过它与内存文件管理表目建立联系，通过该表可以找到保存在外存中的文件。“消息队列指针”指向本进程从其他进程接收到的消息所构成消息队列的链头。“资源使用情况”记载该进程生存期间所使用的系统资源和使用时间，用于记账。对于同一用户来说，他的全部进程所使用的资源都记载在该用户的账目下。“进程队列指针”用于构建进程控制块队列，它是系统管理进程所需要的。

2.2.4 进程的组成与上下文

进程由两个部分组成，即进程控制块和程序，其中程序包括代码和数据等。

1．进程控制块

进程控制块是进程的“灵魂”。由于进程控制块中包含程序的地址信息，通过它可以找到程序在内存或外存储器中的存放地址，也就找到了整个进程。进程控制块存放在系统空间中，只有操作系统才能够对其进行存取，用户程序不能访问。实际上，用户甚至感觉不到进程控制块的存在。

2．程序

程序是进程的“躯体”，其中包括代码和数据两个部分。现代操作系统都支持程序共享的功能，这就要求代码是“纯”的，即在运行期间不会修改自身。另外，在多道程序系统中，内存中同时存在多个程序，这些程序在内存中的存放位置是随机的，而且在运行过程中可能会发生变化，因而代码必须能够浮动，即不采用绝对地址。

数据一般包括静态变量、动态堆和动态栈。堆用来保存动态变量，栈用来保存用户子程序相互调用时的参数、局部变量、返回值、断点等。数据一般是进程私用的，当然也有的系统可以提供进程间共享数据的功能，以实现进程间的信息交换。

应当指出，程序虽然属于进程空间，操作系统程序必须能够访问它，如进程执行输出操作时，通过系统调用进入系统，由操作系统将待输出的数据由进程空间取出并送给指定的外围设备。

进程有两种表示方法，如图 2-5 所示。图 2-5（a）将代码和数据看作一个整体，图 2-5（b）则强调代码部分的可共享性。再次提醒读者注意，进程控制块属于操作系统空间，而程序则属于用户空间。

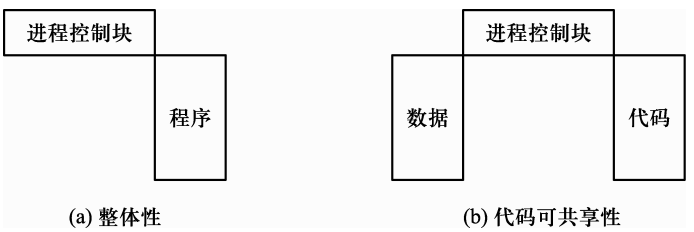

图 2-5 进程的表示

定义 2-3 进程的程序（代码和数据）被称为进程映像（process image）。

进程是在操作系统的支持下运行的。进程运行时操作系统需要为其设置相应的运行环境，如系统堆栈、地址映射寄存器、打开文件表、PSW 与 PC、通用寄存器等。在 UNIX System V 中，将进程的物理实体与支持进程运行的物理环境合称为进程上下文（process context），进程切换（process switch）过程就是进程上下文切换（context switch）的过程。由于进程上下文涉及的内容较多，进程切换一般需要一定的时间才能完成，这是系统为实现并发而付出的额外代价，属于系统开销的一部分。

定义 2-4 系统开销（system overhead）一般是指运行操作系统程序、对系统进行管理所花费的时间和空间。

2.2.5 进程的队列

为实现对进程的管理，系统需要按照某种策略将进程组织成若干队列。由于进程控制块是进程的代表，因而进程队列实际上是由进程控制块构成的队列。因为该队列通常是以链的形式实现的，所以也称为 PCB 链。该链既可以是单向的，也可以是双向的，依照使用方法而定。图 2-6 给出了单向 PCB 链的例子。

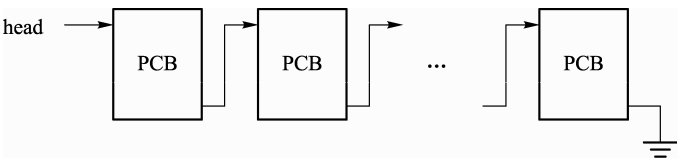

图 2-6 单向 PCB 链

需要注意的是，这里虽然用了“队列”这一术语，但是 PCB 入队列和出队列不一定完全按照先进先出的次序，这与系统对进程的管理策略有关。通常，系统中的进程队列分为以下 3 类。

1．就绪队列

一般整个系统中有一个就绪队列。所有处于就绪态的进程按照某种组织方式排在这一队列中，进程入队列和出队列的次序与处理器调度算法有关。在某些系统中，就绪队列可能有多个，用以对就绪进程进行分类，以方便某种调度策略的实施。

2．等待队列

每个等待事件有一个等待队列。当进程等待某一事件发生时，进入与该事件相关的等待队列中；当某事件发生时，与该事件有关的一个或多个进程离开相应的等待队列，进入就绪队列。

3．运行队列

在单处理器系统中只有一个运行队列，在多处理器系统中每个 CPU 各有一个运行队列，每个队列中只有一个进程。指向运行队列头部的指针被称为运行指示字。

进程在不同队列中的变化情况如图 2-7 所示。进程初创后进入就绪队列；CPU 空闲下来时从就绪队列中选取进程使其运行，获得处理器的进程用完所分得的时间片后回到就绪队列；运

行进程因某一事件受阻进入相应的等待队列，等待进程在所等事件发生后进入就绪队列，运行进程正常结束后离开系统。

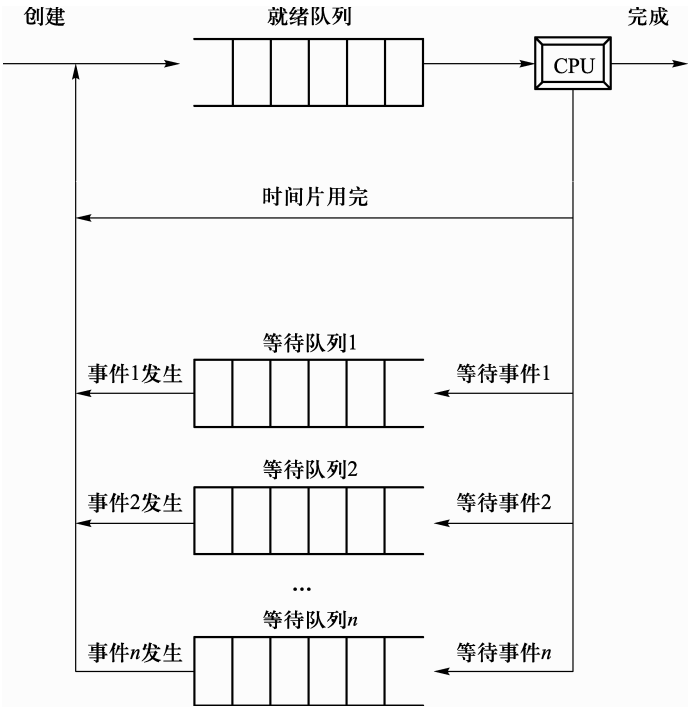

图 2-7 进程队列模型

2.2.6 进程的类型和特性


进程的类型

从操作系统的角度来看，可以将进程分为系统进程和用户进程两大类。

1．系统进程

这类进程属于操作系统的一部分，它们运行操作系统程序，完成操作系统的某些功能，也被称为守护（daemon）进程。一个系统进程所完成的任务是相对独立和具体的，而且在进程的生存周期内保持不变，因而它们通常对应一个无限循环程序，在系统启动后便一直存在，直到系统关闭。现代操作系统内设很多系统进程，完成不同的系统管理功能。系统进程运行于管态，可以执行包括特权指令在内的所有机器指令。由于系统进程承担系统的管理和维护性任务，它们的优先级别通常高于一般用户进程的优先级别。

2．用户进程

这类进程运行用户程序，直接为用户服务。应当指出，所谓“用户程序”，不一定是用户自己编写的程序，例如用户在编译一个 C 程序时，他需要运行 C 语言的编译程序，该程序在目

态运行，但并不是用户自己编写的。也就是说，在操作系统之上运行的所有应用程序都被称为用户进程。

无论系统进程还是用户进程，都具有如下特性。

① 并发性。可以与其他进程一道在宏观上同时向前推进。

② 动态性。进程是执行中的程序。此外，进程的动态性还体现在如下两个方面。首先，进程是动态产生、动态消亡的；其次，在进程的生存周期内，其状态处于经常性的动态变化之中。

③ 独立性。进程是调度的基本单位，它可以获得处理器并参与并发执行。

④ 交互性。进程在运行过程中可能会与其他进程发生直接或间接的相互作用。

⑤ 异步性。每个进程都以其相对独立、不可预知的速度向前推进。

⑥ 结构性。每个进程都有一个进程控制块。

2.2.7 进程间的相互联系与相互作用

多道程序系统中同时运行的并发进程一般有多个，在逻辑上，这些进程之间既可能存在某种联系，也可能相对独立。

① 相关进程。在逻辑上具有某种联系的进程称为相关进程。例如，进程 P_0 在运行过程中创建了两个子进程 P_1 和 P_2，进程 P_1 所产生的输出作为进程 P_2 的输入，则 P_1 和 P_2 是相关进程。此外，进程 P_1 和 P_2 与进程 P_0 之间存在“父子”关系，因而它们都是相关进程。一般来说，属于同一进程家族内的所有进程都是相关的。

② 无关进程。在逻辑上没有任何联系的进程称为无关进程，例如，对于两个相互之间没有交往的用户来说，其进程是无关的。

并发进程之间存在相互制约的关系，这种相互制约的关系称为进程间的相互作用。进程间相互作用的方式有两种，即直接相互作用和间接相互作用。

① 直接相互作用。进程之间不需要通过媒介而发生的相互作用，这种相互作用通常是有意识的。例如，进程 P_1 将一个消息发送给进程 P_2，进程 P_1 的某一步骤 S_1 需要在进程 P_2 的某一步骤 S_2 执行完毕之后才能继续等。直接相互作用只发生在相关进程之间。

② 间接相互作用。进程之间需要通过某种媒介而发生的相互作用，这种相互作用通常是无意识的。例如，进程 P_1 欲使用打印机，该设备当前被另一进程 P_2 所占用，此时进程 P_1 只好等待，以后进程 P_2 用完并释放该设备时，将进程 P_1 唤醒。间接相互作用可能发生在任意进程之间。


进程创建与撤销

2.2.8 进程的创建、撤销与汇聚

除一些系统进程之外，用户进程都是通过系统调用来创建的。一般至少包括两个相关的系统调用，一个用于创建进程，另一个用于撤销进程。这两种命令在不同系统中的实现各异。在 UNIX 系统中的命令格式如下。

1．进程创建

创建进程命令格式为

pid = fork()

创建子进程所要完成的主要工作如下：建立一个进程控制块，并对其内容进行初始化；为该进程分配必要的存储空间，并加载所要执行的程序（在 UNIX 系统中需要通过另一个系统调用 execl 实现）；将进程控制块送入就绪队列。

2．进程撤销

撤销进程命令格式为

exit(status)

完成使命的进程需要终止自己并告知操作系统，这在 UNIX 系统中是通过 exit 系统调用命令实现的。exit 命令的执行将进入操作系统，操作系统将对进程进行善后处理（收集进程状态信息、通知其父进程等），之后将收回进程所占有的所有资源（打开文件、内存等），最后撤销其进程控制块。

除正常终止外，地址越界、非法指令、来自用户或父进程的 kill 信号等原因也可能导致进程的非正常终止。非正常终止的进程也将进入操作系统进行善后处理。

3．进程汇聚

在 Java 语言中，父线程创建子线程之后，可以执行 join 系统调用，等待子线程结束并与其汇聚，之后父线程继续执行。

体现生灭过程的进程状态转换图如图 2-8 所示。注意创建与结束属于操作系统中的系统调用，而其他状态转换条件对应于操作系统核心中的内部函数。

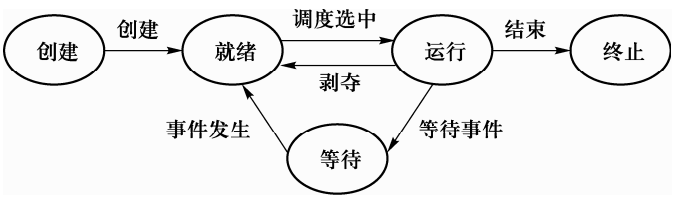

图 2-8 体现生灭过程的进程状态转换图

2.2.9 进程与程序的联系和差别

进程和程序是既有联系，又有差别的两个概念，读者应当注意加以区分。

1．进程与程序的联系

程序是构成进程的组成部分之一，一个进程存在的目的就是执行其所对应的程序。如果没有程序，进程就失去了其存在的意义。

2．进程与程序的差别

① 程序是静态的，而进程则是动态的。

② 程序可以写在纸上或在某种存储介质上长期保存，而进程具有生存周期，创建后存在，撤销后消亡。

③ 一个程序可以对应多个进程，但是一个进程只能对应一个程序。例如，一组学生在一个分时系统中做 C 语言实习，他们都需要使用 C 语言的编译程序对其源程序进行编译，为此每个学生都需要一个进程，这些进程都运行 C 语言的编译程序。另外，一个程序的多次执行也分别对应不同的进程。

2.3 线程与轻进程

2.3.1 线程的引入


线程及其控制块

早期的操作系统是基于进程的，一个进程中只包含一个执行流，进程是处理器调度的基本单位。当处理器由一个进程切换到另一个进程时，整个上下文都要发生变化，系统开销较大，显得笨重，相关进程间的耦合度差。

在许多应用中，一些执行流之间具有内在的逻辑关系，涉及相同的代码或数据。如果将这些执行流放在同一个进程的框架下，则这些执行流之间的切换便不涉及地址空间的变化，这就是线程思想的由来。

同一进程中的多个线程既可以执行相同的代码段，对应同一服务的多个请求；也可以执行不同的代码段，体现逻辑上的合作关系，这些合作线程可以利用共享的数据成分相互交往。

2.3.2 线程的概念

定义 2-5 线程（thread）又称轻进程（light weight process，LWP），是进程内的一个相对独立的执行流。

一个进程可以包含多个线程，这些线程执行同一程序中的相同代码段或不同代码段，共享数据区和堆。一般认为，进程是资源的分配单位，线程是 CPU 的调度单位。

与进程相比，线程具有如下优点。

① 上下文切换速度快。由同一进程中的一个线程切换到另一个线程只需改变寄存器和栈，包括程序和数据在内的地址空间不变。

② 系统开销小。创建线程比创建进程所需完成的工作少，因而对于客户请求，服务器动态创建线程比动态创建进程具有更高的响应速度。

③ 通信容易。由于同一进程中的多个线程的地址空间共享，一个线程写到数据空间的信息可以直接被该进程中的另一线程读取，方便快捷。

2.3.3 线程的结构

图 2-9 给出多进程结构。如果这两个进程具有一定的逻辑联系，比如二者是执行相同代码

的服务程序，或者二者为协同进程，则可以用多线程结构实现，如图 2-10 所示。

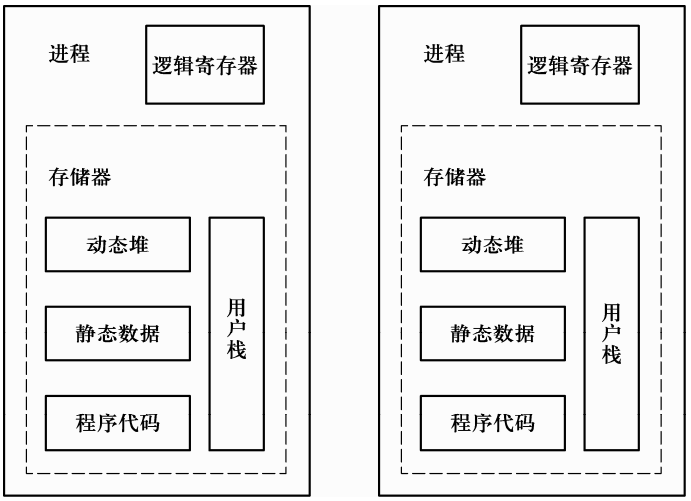

图 2-9 多进程结构

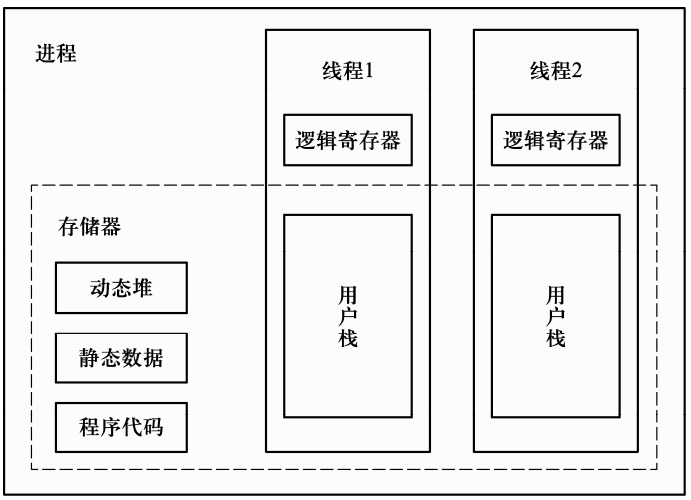

图 2-10 多线程结构

注意图 2-9 和 2-10 中用了“逻辑寄存器”一词，是想强调在单 CPU 系统中，硬件寄存器只有一套，哪个进程（线程）运行，哪个进程（线程）使用，可以理解为每个进程有一组虚拟的寄存器。由于线程是一个执行流，每个线程运行时，都涉及函数的调用和返回，而且函数调用可以嵌套，因而需要各自独立的栈，用以保存函数调用的返回点、参数、局部变量、返回值。另外图中只画出了用户空间的内容，未涉及操作系统空间的控制信息，将在后面详述。

2.3.4 线程控制块

与进程相似，线程也是并发执行的，即时断时续的。为此需要一个类似于 PCB 的数据结构以保存现场等控制信息，这个控制结构称为线程控制块（thread control block，TCB）。

定义 2-6 线程控制块是标志线程存在的数据结构，其中包含系统对线程进行管理所需要的全部信息。

不过一般线程控制块中的内容较少，因为有关资源分配等信息已经记录于所属进程的进程控制块中。线程控制块中的主要信息如图 2-11 所示，其中线程状态及其转换情况与进程状态及其转换情况相似，现场信息主要包括通用寄存器、指令计数器以及用户栈指针。对于操作系统支持的线程，线程控制块中还应包含系统栈指针。

线程控制块可能属于操作系统空间，也可能属于用户进程空间（由运行系统管理和使用），这取决于线程的实现方式。

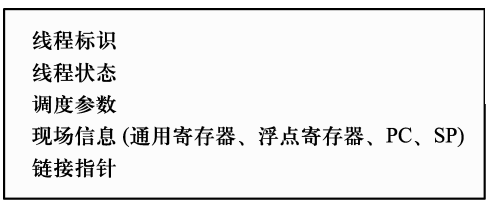

图 2-11 线程控制块

2.3.5 线程的实现


线程的实现

线程有两种实现方式：在目态实现的用户级别线程，在管态实现的核心级别线程。Sun Solaris 系统采用了混合形式。

1. 用户级别线程

早期的线程都是用户级别线程（user level thread，ULT），由系统库支持。线程的创建和撤销，以及线程状态的变化都由库函数控制并在目态完成，如图 2-12 所示。与线程相关的控制结构——线程控制块保存在目态空间中并由运行系统维护。由于线程对操作系统不可见，系统调度仍以进程为单位，核心栈的个数与进程个数相对应。

对于用户级别线程，若同一进程中的多个线程中至少有一个处于运行态，则该进程的状态为运行态；若同一进程中的多个线程均不处于运行态，但是至少有一个线程处于就绪态，则该进程的状态为就绪态；若同一进程中的多个线程均处于等待态，则该进程的状态为等待态。

用户级别线程的优点在于：线程不依赖于操作系统，可以采用与问题相关的调度策略，灵活性好；同一进程中的线程切换不需要进入操作系统，因而实现效率较高。用户级别线程的缺点在于：同一进程中的多个线程不能真正并行，即使在多处理器环境中；由于线程对操作系统不可见，调度在进程级别，某进程中的一个线程通过系统调用进入操作系统受阻，该进程的其

他线程也不能运行。

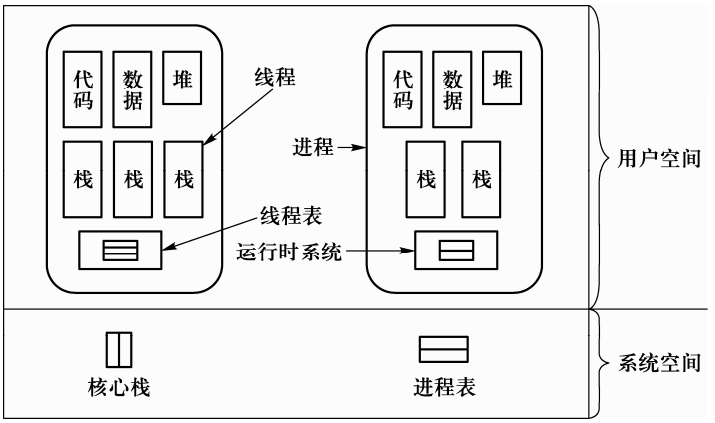

图 2-12 用户级别线程

用户级别线程所具有的优势是核心级别线程所不具备的，因而即使在所有现代操作系统都提供了核心级别线程之后，用户级别线程仍然被多数系统支持。

2．核心级别线程

核心级别线程（kernel level thread，KLT）通过系统调用由操作系统创建，线程的控制结构——线程控制块保存于操作系统空间，线程状态转换由操作系统完成，线程是 CPU 调度的基本单位。另外，由于系统调度以线程为单位，操作系统还需要为每个线程保持一个核心栈，如图 2-13 所示。

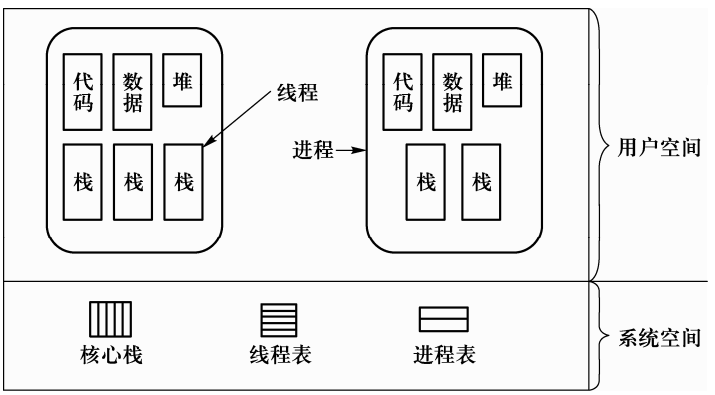

图 2-13 核心级别线程

对于核心级别线程，进程状态不具有实际意义，可以将其省略。

核心级别线程的优点是并发性好，在多处理器环境中同一进程中的多个线程可以真正并行执行。核心级别线程的缺点是线程的控制和状态转换需要进入操作系统完成，系统开销比较大。

3．混合线程

Solaris 中的进程被称为任务（task），为结合用户级别线程与核心级别线程的优点，Solaris 系统采用混合线程（hybrid thread）结构。关于线程有以下 3 种实体与之相对应。

① 用户级别线程（ULT）：是用户程序并发或并行执行的基本单位，每个任务中至少包含一个 ULT，ULT 对用户可见，对操作系统不可见；

② 核心级别线程（KLT）：是操作系统调度执行的基本单位，对操作系统可见，对用户不可见，KLT 与处理器（可多个）之间可以是多对多关系，也可以是一对一关系（如专用处理器）；

③ 轻进程（LWP）：对用户可见，对操作系统也可见，每个进程至少包含一个轻进程，用户级别线程与轻进程之间可以是一对一关系，也可以是多对多关系，只有与轻进程动态绑定的用户级别线程能够与核心通信，轻进程与核心级别线程之间具有一对一关系。

可见，Solaris 系统是通过轻进程把用户级别线程与核心级别线程联系在一起的，那么为什么称之为混合线程呢？

我们来看一下，如果任务中包含多个用户级别线程，但只含有一个轻进程，用户程序动态绑定用户级别线程与轻进程之间的关系，即哪个用户级别线程与轻进程绑定，则哪个用户级别线程可以运行。如果任务中每个用户级别线程都有一个轻进程与之绑定，则这些用户级别线程可以并行执行，即相当于核心级别线程。如果用户级别线程与轻进程具有多对多绑定关系，则对应更一般的线程控制结构，可见 Solaris 线程提供了非常灵活的线程控制模式。

2.3.6 线程的应用

许多计算任务和信息处理任务在逻辑上涉及多个控制流，这些控制流具有内在的并发性。当其中一些控制流被阻塞时，另一些控制流仍可继续。在没有线程支持的条件下，只能采用单进程或多进程模式。单进程不能表达多控制流，多进程开销大而且在无共享存储空间的条件下进程间的交互困难。采用多线程一方面可以提高应用程序的并行性，另一方面也使程序设计简洁、明晰。

例如，考虑 Word 文字处理程序，该程序在运行时一方面需要接收用户输入的信息，另一方面需要对文本进行词法检查，同时需要定时将修改结果保存到临时文件中以防意外事件发生。易见，这个应用程序涉及 3 个相对独立的控制流，这 3 个控制流共享内存缓冲区中的文本信息。以单进程或多进程模式都难以恰当地描述和处理这一问题，而同一进程中的 3 个线程是最恰当的模型。

又如，考虑 Web 服务器的服务模式。一个 Web 服务器可以同时为许多 Web 用户服务，对应每个 Web 请求，Web 服务器将为其建立一个相对独立的控制流。若以进程模式实现，由于开销大将影响响应速度，以线程模式实现则更为便捷。对应每个 Web 请求，系统可以动态弹出（pop up）一个线程。为使响应速度更快，也可以事先将线程建立起来，当请求到来时选派一个服务线程。这些服务线程执行相同的程序，因而对应同一个进程。

通过以上分析可以归纳出引入多线程（multithread）程序设计的原因。

① 某些应用具有内在的多个控制流结构，这些控制流具有合作性质，需要共享内存。采用多线程易于对问题建模，从而得到最自然的解决算法。

② 在需要多控制流的应用中，多线程比多进程在速度上具有更大优势。统计测试结果表明，线程的建立速度比进程的建立速度快 100 倍，进程内线程间的切换速度与进程间的切换速度也有数量级之差。

③ 采用多线程可以提高处理器与设备之间的并行性。在单控制流的情形下，启动设备的进程进入核心后将被阻塞，此时该进程的其他代码也不能执行。若此时无其他可运行程序，处理器将被闲置。多线程结构在一个线程等待时，其他线程可以继续执行，从而使设备和处理器并行工作。

④ 在有多个处理器的硬件环境中，多线程可以并行执行，既可提高资源利用效率，又可提高进程推进速度。

当然应该记住，采用多线程也是有条件的：同一进程中的多个线程具有相同的代码和数据，这些线程之间或者是合作的（执行代码的不同部分），或者是同构的（执行相同的代码）。

2.4 作业

定义 2-7 用户要求计算机系统为其完成的计算任务的集合称为作业（job）。

作业是早期批处理系统引入的一个概念，分时系统用户在一次登录后所进行的交互序列也常被看作一个作业。一般来说，作业是比进程还大的一个概念，一个作业通常包含多个计算步骤，作业中的一个相对独立的处理步骤称为一个作业步（job step）。作业步之间具有顺序或并发关系。一个作业步通常可以由一个进程来完成，这样一个作业在内存处理时通常与多个进程相对应，即作业与进程之间具有一对多的关系。

2.4.1 批处理作业

为实现对作业的管理，需要保持相关的信息，这些信息包括作业名称、作业状态、调度参数、资源需求、相关进程、作业长度、在输入井与输出井中的存放位置、记账信息等，这些信息被保存在一个称为作业控制块（job control block，JCB）的数据结构中。

定义 2-8 作业控制块是标志作业存在的数据结构，其中包含系统对作业进行管理所需要的全部信息。

作业由假脱机输入程序（SPOOL 输入程序）控制进入输入井，经操作系统的作业调度程序选择进入内存，并为其建立作业控制进程。作业控制进程解释作业说明书的语句，根据作业步的要求为其建立进程。

2.4.2 交互式作业

对分时系统来说，通常将分时用户的一次登录称为一个作业。一次登录可以向系统提出多个请求，每个请求都可能对应一个进程，这样分时作业与进程之间也是一对多的关系。

为实现对分时用户的管理，系统设置一个口令文件，在 UNIX 系统中该文件保存在/etc/passwd 中，可称为 passwd 文件。它一般包括注册用户的所有信息，内容如图 2-14 所示。其中用户名和口令用于登录，用户 id 为用户内部标识，用户根目录为保存私用文件和目录的地方。例如在 UNIX 系统中的/usr 目录下，对应用户 zhang 的根目录可能为/usr/zhang。注册资金为预缴的费用，每次登录使用系统后将实际发生的费用从注册资金中扣除。

用户名	口令	用户id	用户根目录	注册资金
…	…	…	…	…

图 2-14 系统 passwd 文件

passwd 文件为系统文件，只有特权用户才能访问它，而一般用户不能访问它。这是系统保护与安全的基本要求。

注册用户可以随时登录系统。登录成功后，系统为该用户创建一个服务进程。对于命令行界面，该进程执行命令解释程序，读入终端命令并解释处理，处理过程可能调用操作系统，甚至可能创建子进程。UNIX 系统 shell（外壳）的工作原理如图 2-15 所示。

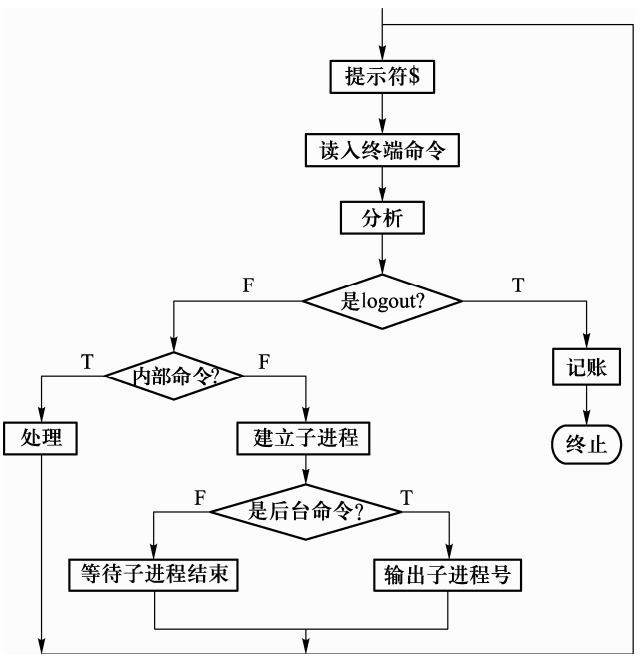

图 2-15 终端命令解释程序工作原理

在 UNIX 系统中以“&”为结束符的终端命令为后台命令，对于后台命令 shell 在输出进程号后便可给出提示符并读取下一个终端命令。这样的进程在后台运行，以后用户可以根据进程号检查后台进程的运行状况。容易看出，在分时系统中，对于一个终端用户，在内存中可以同时有多个进程为其服务。考虑处理器是按进程（线程）分配的，不同终端实际获得 CPU 的时间一般是不同的。

2.5 系统举例

2.5.1 Java 线程

Java 线程可以通过扩展 thread 类来实现，举例如下。

```
Class worker extends thread
{
    public void run(){
        system.out.println("I’m a worker thread");
    }
}
public class first
{
    public static void main（string args[]）{
        worker runner = new worker();
        runner.start();
        system.out.println("I’m the main thread");
    }
}
```

Java 提供如下有关线程的 API 函数：suspend()用于挂起当前线程，sleep()用于为当前线程指定睡眠时间，resume()用于唤醒某个被挂起的线程，stop()用于终止某一线程。

Java 线程有如下 7 种基本状态。

① 初始状态。新建的线程，父线程执行 New 命令创建子线程。

② 可运行（runnable）状态。初始状态的线程执行 start 命令，进入可运行状态。

③ 运行中状态。线程获得处理器运行。

④ 阻塞状态。等待某一事件发生。

⑤ 入口集状态。线程调用某一对象的同步方法，因互斥要求进入该对象的入口集。

⑥ 等待集状态。线程在对象的同步方法中执行 wait()标准方法，释放该对象的互斥锁，允许入口集任一线程进入该对象，在该对象中执行 notify()或 notifyAll()标准方法时，等待集中任

一线程或全部线程转移到入口集。

⑦ 结束状态。

各状态转换条件如图 2-16 所示。

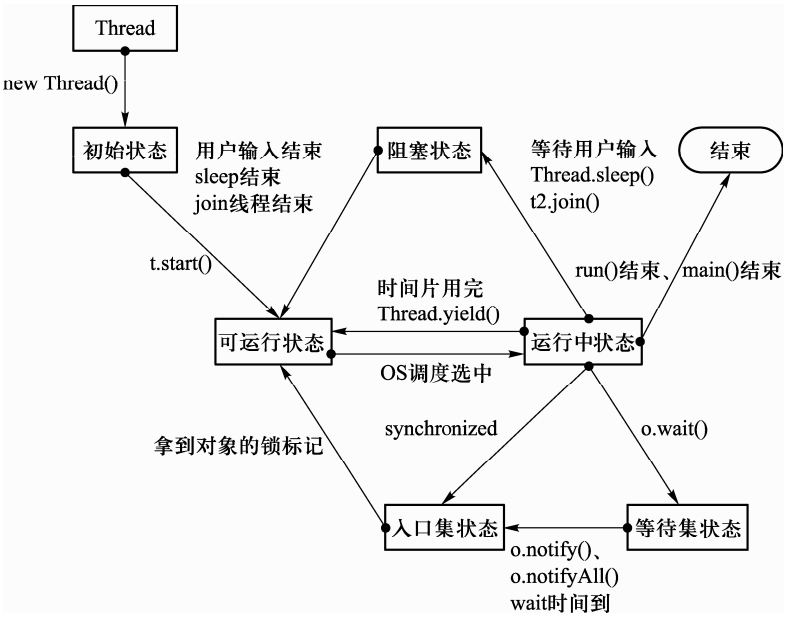

图 2-16 Java 线程状态转换图

Java 线程是由 Java 虚拟机（JVM）支持的，JVM 位于操作系统之上。Java 线程与操作系统线程之间的对应关系由 JVM 确定。对于 Windows NT 的 JVM，Java 线程与操作系统线程具有一对一关系，对于 Solaris 的 JVM，其对应关系为多对多。

2.5.2 Linux 进程与线程

在 Linux 系统中，进程与线程在系统内部具有统一的表示：线程是与其父进程具有相同地址空间的进程。进程与线程的差别通过与 fork 不同的另一个系统调用 clone 体现出来。fork 用于为子进程创建一个具有全新上下文的独立地址空间。而 clone 并不产生新的地址空间，子进程共享父进程的地址空间。clone 为应用程序提供细粒度的共享成分控制。

使用 clone 为进程（线程）间通信提供了便利条件，但是必须考虑共享成分的互斥与并发控制问题。clone 系统调用的形式如下：

pid = clone（function,stack_ptr,sharing_flag,arg）

其中，function 为新线程执行代码；stack_ptr 为新线程栈指针；sharing_flag 为共享标志位，包括 CLONE_VM、CLONE_FS、CLONE_FILES、CLONE_SIGHAND、CLONE_PID；arg 为调用参数。

2.5.3 Windows 10 的进程、线程与纤程

1．进程

进程是一种活动对象，进程管理器提供进程创建、撤销等功能。

在 Win32 环境中创建进程的过程如下：当 Win32 应用执行 CreateProcess 调用时，消息被发送给 Win32 子系统，后者调用进程管理器创建进程，进程管理器调用 OM 创建进程对象，然后将对象句柄返回给 Win32。Win32 子系统再次调用进程管理器为该进程创建线程，最后 Win32 将句柄返回给新进程和线程。

2．线程

在 Windows 10 中，线程也被视为对象加以管理，每个进程中至少包含一个线程，线程是核心调度的基本单位。每个线程包括两个栈：系统栈和用户栈。线程通过 Win32 调用创建，可以访问进程内的对象。当进程中的所有线程终止时，进程终止。

Windows 线程包含 7 个状态，各个状态及其含义解释如下。

① 初始（initialized）：初始创建过程所经历的状态。

② 就绪（ready）：等待在任何处理器上运行。

③ 备用（standby）：已经选择好线程的执行处理器，正在等待切换以进入运行状态。每个处理器上只有一个处于备用状态的线程。

④ 运行（running）：已经完成描述表切换，在某个处理器上运行，直到被更高优先级的线程剥夺，或者时间配额用完，或者终止，或者因资源进入等待状态。

⑤ 等待（waiting）：等待某对象，以同步线程的执行。进入该状态的原因：主动等待对象同步其操作，操作系统可以代替线程等待，环境子系统可以指示线程等待。

⑥ 过渡（transition）：与就绪态类似，但是其核心堆栈位于外存储器。

⑦ 终止（terminated）：执行结束。线程对象可能会被删除，也可能会被重新初始化，并被再次使用。

还有未知（unknown）状态，即系统不能确定线程的状态，这经常是因为错误导致的。

Windows 10 的线程状态转换如图 2-17 所示。

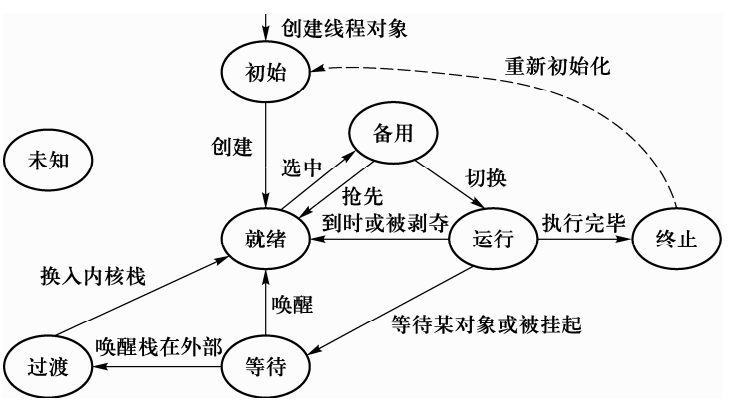

图 2-17 Windows 10 的线程状态转换

除用户线程外，操作系统核心拥有一组守护线程（daemon thread），负责系统管理。例如，“赃页刷新线程”负责将内存中修改过的页面及时写回到磁盘，一方面减少了淘汰时写回操作，另一方面也提高了可靠性。另外，许多系统服务功能也是由系统守护线程提供的。

3．纤程

在 Windows 10 中，纤程（fiber）相当于用户级别线程或轻进程。纤程由 Win32 库函数支持，对核心是不可见的。纤程可以通过 SwitchToFiber 显式切换至另一个合作纤程，以实现合作纤程之间的协同。

纤程包含独立的目态栈、寄存器状态等控制信息。目态控制的纤程转接要求较高的编程经验。由于纤程属于目态对象，一个纤程被阻塞意味着所在线程被阻塞。应用程序可以通过 ConvertThreadToFiber 将线程转换为纤程。与线程相比，纤程具有切换速度快的特点。

习 题 二

1. 为何引入多道程序设计？在多道程序系统中，内存中作业的道数是否越多越好？请说明原因。

2. 多道程序设计会带来哪些问题？如何解决？

3. 什么是进程？进程具有哪些主要特性？试比较进程与程序之间的相同点与不同点。

4. 有人说，用户进程所执行的程序一定是用户自己编写的。这种说法对吗？如果不对，试举例说明之。

5. 什么是进程上下文？进程上下文包括哪些成分？哪些成分对目态程序是可见的？

6. 进程一般具有哪 3 个主要状态？举例说明状态转换的原因。

7. 对于循环轮转进程调度算法、可抢占 CPU 的优先数进程调度算法、不可抢占 CPU 的优先数进程调度算法，分别画出进程状态转换图。

8. 有几种类型的进程队列？每类各应设置几个队列？

9. 什么是进程控制块？进程控制块中一般包含哪些内容？

10. 什么是线程？图示进程与线程之间的关系。

11. 试比较进程状态与该进程内部线程状态之间的关系。

12. 什么是线程控制块？线程控制块中一般包含哪些内容？

13. 同一进程中的多个线程有哪些成分是共用的，哪些成分是私用的？

14. 试比较用户级别线程与核心级别线程间在以下几个方面的差别和各自的优、缺点。

（1）创建速度 （2）切换速度 （3）并行性 （4）线程控制块的存储位置

15. 试比较 Linux 系统中 fork()和 clone()两个系统调用之间的差异。

16. 何谓作业？何谓作业步？作业何时转变为进程？

17. 试分析作业、进程、线程三者之间的关系。

18. 何谓系统开销？试举 3 个例子说明之。