第3章

chapter 3

进 程 管 理

第 1 章已介绍了多道程序设计改善了 CPU、存储器和外设等系统资源的使用情况，多道程序设计是操作系统最基本、最重要的概念。

3.1 多道程序设计

衡量一个系统效率的一个指标就是吞吐率，其定义为单位时间内系统所处理作业（程序）的道数，即

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

显然，系统效率与系统资源的利用率密切相关，其主要涉及处理机、存储器、设备这样一些硬件资源的利用率问题。如果系统资源利用率高，则单位时间内完成的有效工作多；反之，系统资源利用率低，则单位时间内完成的有效工作就少，吞吐量小。所以，提高系统的吞吐量应当提高系统资源的利用率。下面通过例子说明处理机利用率与吞吐率的情况，如图 3.1 所示。

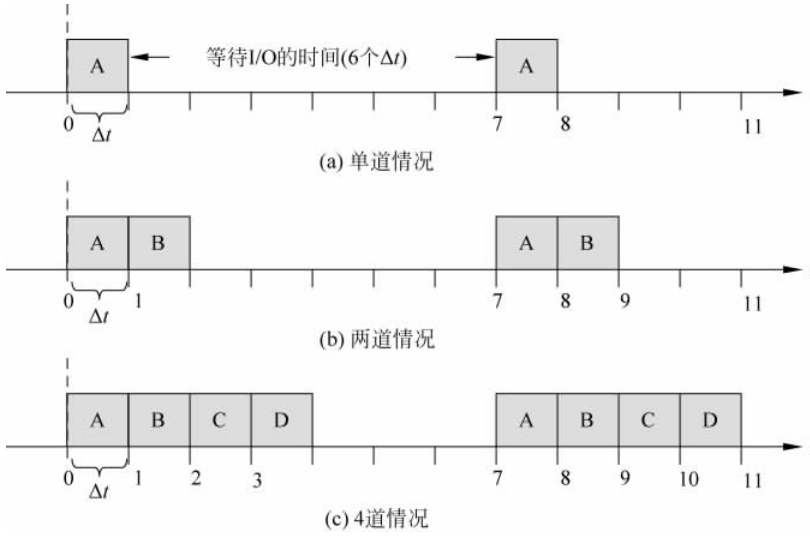


图 3.1 作业道数分别为单道、两道和 4 道情况

其中，A、B、C、D分别表示用户程序。为简单起见，忽略外部设备资源的利用情况，并假定4个程序分别都需要运行2个 Δt 时间才能完成相应的任务，又在每一个程序运行一个 Δt 时间之后，都需要进行6个 Δt 时间的I/O操作，再运行一个 Δt 时间完成。它们的吞吐率分别为1/8=0.125、2/9=0.222和4/11=0.363。4道作业的情况比单道提高了近3倍。显然内存存放多道程序不仅使得内存空间得以充分利用，而且带来了处理机利用率的提高，从而使整个系统效率得以提高。

由图3.1可知在多道程序设计的机制下，内存和处理机的利用率得到了显著的提高。问题是内存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同时处理的程序的数目。其次，物理设备的数量也是一个制约条件，如果内存中可同时运行的程序过多，这些程序之间可能会因为相互等待被其他程序占用的设备资源（如I/O设备），反而可能会影响系统效率。当然，处理机的竞争在作业道数过多的情况下更加激烈，可能会产生两个不利后果。一是影响系统的响应速度，二是产生过多的系统开销（系统本身需要运行必要的程序进行相应的控制和管理）。一般来说，系统同时接纳用户程序的数目与系统的功能和系统配置有关。

正如前所述，多道程序设计在提高系统效率的同时，也加剧了系统资源的竞争，因此要协调程序与资源的关系。处理机、存储器和外部设备是计算机系统中重要的硬件资源，因而需要解决处理机资源管理、存储器分配和回收以及外部设备资源管理等问题。

3.2 进程的概念

在计算机和操作系统的发展过程中，人们在不断地创新和总结经验，产生了许多具有指导意义的重要思想和理论。不论是计算机硬件材料以及性能的增强，还是用户对更加简洁的计算机操作方式的期望，都要求操作系统软件必须不断地推陈出新。所以，回顾操作系统的发展历程，总结那些成功的大型系统软件的开发经验，同时也对导致失败的原因进行分析，从而提炼出了对编制各类系统软件的思想和理论。

多道程序设计是操作系统最基本的思想，然而系统如何协调各个程序并发运行、资源共享，如何刻画程序可以运行的系统环境就需要一种指导思想、一种机制来实现。如上一节多道程序运行的简单情况，在图3.1中，A、B、C、D是根据什么原则和依据运行的，其中每一个程序在系统中当前的可感知的信息有哪些、是什么、怎样体现它们“走走停停”的活动等，这就需要反映程序的一种动态性，而在程序的静态概念下是无法刻画系统中的并发特性的。由于各程序同时存在于主存中，它们之间必定会存在相互依赖，相互制约的关系（亦即间接制约关系、直接制约关系），这也是程序本身无法反映的。

在第1章中，关于操作系统的各种观点之一——虚拟机的观点对于提升计算机整体性能，尤其在单处理机系统中是十分重要的思想。虚拟化思想的本质就是整合计算机系统的软硬资源优势，力争全面提高为用户服务的水平。虚拟化思想是计算机技术面向应用的最高境界。如果说虚拟存储器是通过虚拟存储技术，将系统中两种异构的存储设备（主存和磁盘）映射为一个单一的、有效的存储资源，达到对用户完全透明的、方便使用的目的话，虚拟主机就是将系统内有限的处理机通过对用户程序的管理和调度机制扩充为多个虚拟机，使其同时为多于处理机个数的用户服务。

现代操作系统的重要特点是程序的并发执行，系统所拥有的资源被共享和系统的用户随机地使用。这三个特点是互相联系和互相依赖的，它们是互相独立的用户如何使用有限的计算机系统资源的反映。通常，操作系统的重要任务之一是使用户充分、有效地利用系统资源。采用一个什么样的概念来描述计算机程序的执行过程和作为资源分配的基本单位才能充分反映操作系统的并发执行、资源共享及用户随机的特点，这个概念就是进程。

3.2.1 前驱图和程序执行

1. 前驱图的定义

前驱图(Procedence Graph)是有向无环图 DAG(Directed Acyclic Graph)，是用来反映和研究系统内所发生的事件之间的关系。图中的每个结点可用于表示一条语句、一个程序段或进程。结点间的有向边则表示在两结点之间存在的偏序或前驱关系“→”，$\rightarrow=\{(p_i,p_j)\mid p_i$ 必须在 p_j 开始之前完成$\}$。如果$(p_i,p_j)\in\rightarrow$，可写成 $p_i\rightarrow p_j$，称 p_i 是 p_j 的前驱，而 p_j 是 p_i 的直接后继。在前驱图中，没有前驱的结点为初始结点，没有后继的结点为终止结点。图 3.2 给出了 7 个结点的前驱图。在图 3.2 中，存在下面的前驱关系：

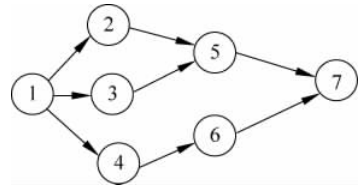


图 3.2 具有 7 个结点的前驱图

$$P_1\rightarrow P_2,P_1\rightarrow P_3,P_1\rightarrow P_4,P_2\rightarrow P_5,P_3\rightarrow P_5,P_5\rightarrow P_7,P_4\rightarrow P_6,P_6\rightarrow P_7$$

或表示为：

$$P=\{P_1,P_2,P_3,P_4,P_5,P_6,P_7\}$$

$$\rightarrow=\{(P_1,P_2),(P_1,P_3),(P_1,P_4),(P_2,P_5),(P_3,P_5),(P_4,P_6),(P_5,P_7),(P_6,P_7)\}$$

2. 程序的顺序执行

程序是指令(或语句)的集合，是一个在时间上按严格次序前后、相继的操作序列。而指令之间是顺序关系，是一个静态的概念。程序体现了编程人员要求计算机完成所要求功能时所应该采取的顺序步骤。显然，一个程序只有经过执行才能得到最终结果，且一般用户在编写程序时不考虑在自己的程序执行过程中还有其他用户程序存在这一事实。另外，计算机 CPU 是通过时序脉冲来控制顺序执行指令的。其执行过程可以描述为：

```
Repeat   IR ← M[pc]
         pc ← pc + 1
        执行 IR 中的指令
Until CPU halt
```

其中，IR 为指令寄存器，pc 为程序计数器，M 为存储器。显然，程序的顺序性与计算机硬件的顺序性是一致的。这里把一个具有独立功能的程序独占处理机直至最终结束的过程称为程序的顺序执行。

一般来说，一个程序由若干个语句或程序段组成，而程序在执行时，总是需要输入的部分，然后进行计算，最后给出结果。为了描述的方便，假定用 I、C 和 P 来抽象地分别表示输入、计算和输出(打印)操作(也可以为程序中的多条顺序语句)，分别需要占用系统输入资源、处理机资源及输出资源。对任意的用户程序，可以有图 3.3 的前驱图。

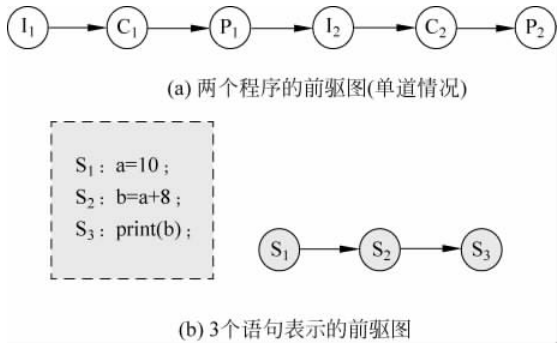


图 3.3 程序的前驱图

图 3.3(a)和图 3.3(b)分别表示了两个程序执行的前驱图和一个程序中 3 个语句所表示的前驱图。在图 3.3(b)中，I、C 和 P 结点分别表示 3 个语句，其中的关系也是显然的，即 b 的计算必须在 a 赋值之后才能计算，同理，b 的输出也一定是在 b 计算完成之后。这种抽象的表示是在说明程序执行的关系和顺序性。如果考虑时间上的关系，可有下面的示例，如图 3.4 所示。

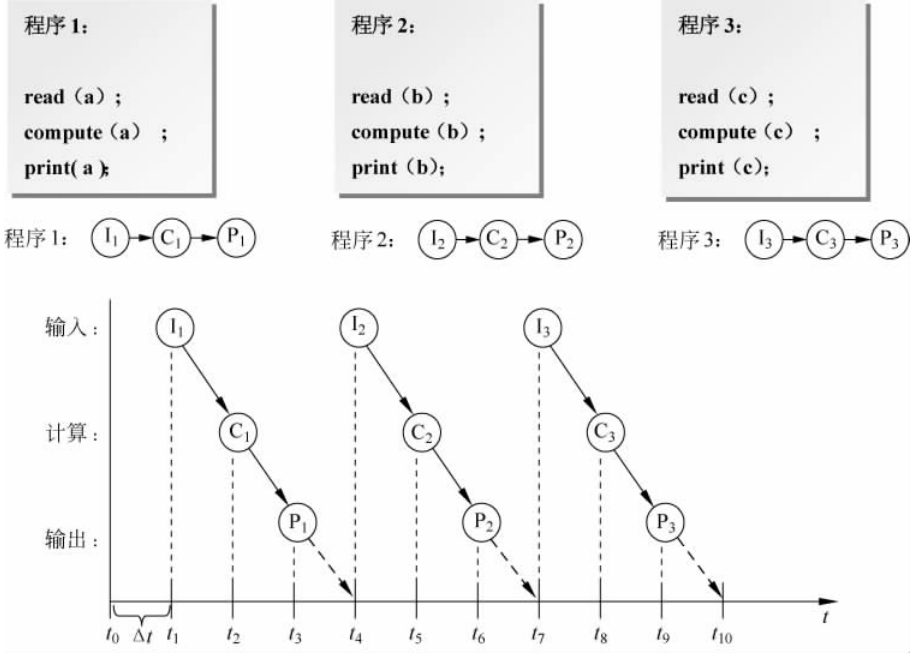


图 3.4 3 个程序间顺序执行

为讨论方便，假定任意的一个语句(操作)的执行时间为一个 Δt 时间，显然，3 个程序之间的顺序执行所需的时间为 9 个 Δt。分别在 t_4、t_7、t_{10}得出计算结果。这种程序的顺序执行具有如下特点。

(1) 顺序性。程序顺序执行时，其执行过程可看作一系列严格按程序规定的状态转移过程。

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

（3）可再现性。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。

程序顺序执行的特性可为程序员检测和校正程序的错误带来很大的方便。

3. 多道程序系统中程序执行环境的变化

在许多情况下，需要计算机能够同时处理多个具有独立功能的程序。批处理系统、分时系统、实时系统等都是这样的系统，如图 3.1 所示。这样的执行环境具有如下 3 个特点。

（1）独立性。每道程序都是逻辑上独立的，它们之间不存在逻辑上的制约关系。

（2）随机性。在多道程序环境下，特别是在多用户环境下，程序和数据的输入与执行开始时间都是随机的。

（3）资源共享。资源共享将导致对程序执行速度的制约。输入输出设备数量有限将导致这些设备被共享、内存有限将导致内存被共享。对于单 CPU 系统，如果有 $N(N\geqslant 1)$道程序，由于共享将导致 $N-1$ 道程序处在等待 CPU 的状态。

如图 3.4 所示，对于任意一个程序，存在着 $I_i\rightarrow C_i\rightarrow P_i(i=1,2,3)$这样的前驱关系，因而对一个用户程序的输入、计算和打印这 3 个操作，必须顺序执行，但在多道环境下，并不存在，或并不要求 $P_i\rightarrow I_{i+1}$关系，即 I_i、C_j 和 $P_k(i\neq j\neq k)$之间并不存在前驱关系，因而在对一批程序处理时，可使它们并发执行。例如，当一个程序的输入完成后，所需要的输入设备释放，继而开始进行计算，在其进行计算的时候，另一个程序的输入就可以同时进行，即 C_i 与 I_{i+1}可同时进行操作，这就产生了并发操作，见图 3.5 所示。

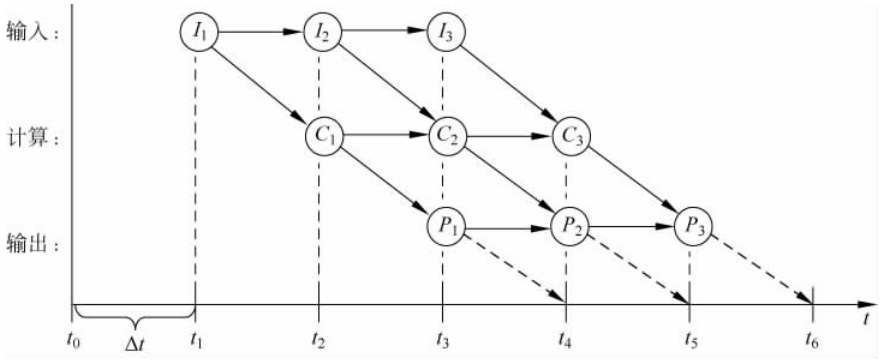


图 3.5　3 个程序并发执行的前驱图

由图 3.5 可以看出存在如下的前驱关系：

（1）对每一个程序，依然保持 $I_i\rightarrow C_i\rightarrow P_i(i=1,2,3)$，即每一个程序内语句的顺序性，是程序本身的内在逻辑性，是用户的前驱关系。

（2）对于不同程序之间，存在 $I_i\rightarrow I_{i+1}$，$C_i\rightarrow C_{i+1}$，$P_i\rightarrow P_{i+1}$，表明由于系统资源的竞争带来的顺序性，即任何一个程序操作步所需的系统资源应在另一个程序使用完毕后才能开始，这种顺序性是系统资源的前驱关系，非程序逻辑关系。

（3）对于不同程序之间的 I_{i+2}、C_{i+1}和 P_i，由图 3.5 可以看出它们之间不存在顺序性，即没有前驱关系，亦非程序逻辑关系。这说明它们之间是可以并发执行的，在时间上有重

叠，这是系统的并发性，它带来了 3 个程序完成的时间的缩短，由原来的 9 个 Δt（参见图 3.4）减少为 5 个 Δt，提高了(9－5)/9×100% ＝ 44%。

应当说明的是，抛开典型的 I、C、P 关系，对于一个程序内部的语句之间在并发环境下，依然可有并发执行的情况存在，如有下面的由 4 条语句构成的程序段。

```
S0: x = a + 10;
S1: y = b - a;
S2: z = x + y - 10;
S3: print(z);
```

由图 3.6 可以看出，S_2 必须在 x 和 y 有值之后才能执行；S_3 也必须在 z 计算完成之后才能进行输出；而 S_0 和 S_1 可以并发执行，它们之间不存在依赖关系。

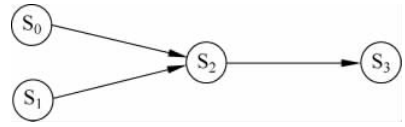


图 3.6 4 条语句的前驱图

这说明程序内部的各程序段(块)之间是否具备并发执行，是由其之间的依赖关系所决定的，这种关于程序并行性的挖掘是属于程序并行研究的课题，已超出本书所讨论的范畴，故不在此加以深入讨论。

4. 程序的并发执行

操作系统的一个重要特征就是并发性，它是增强计算机系统的处理能力和提高资源利用率所采取的一种技术。程序的并发执行通过前面的分析可分为两种。

(1) 多道程序系统的程序执行环境变化所引起的多道程序的并发执行。由于资源的有限性，多道程序的并发执行总是伴随着资源的共享与竞争，从而制约各道程序的执行速度。显然无法做到在微观上，也就是在指令级上的同时执行。因此，尽管多道程序的并发执行在宏观上是同时进行的，但在微观上仍是顺序执行的。

(2) 并发执行是在某道程序的几个程序段中，包含着一部分可以同时执行或顺序颠倒执行的代码。

在时间上来表示，并发执行是一个程序的开始是在另一个程序结束之前。如果忽略程序运行之间的“走走停停”的中间过程，而仅考察其开始和结束点，可由图 3.7 表示。

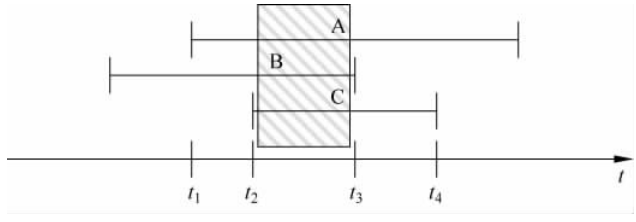


图 3.7 3 个程序在执行时间上的重叠

在图 3.7 中，程序 A、B 和 C 在 t_2 和 t_3 上重叠，A 和 B 在 t_1 和 t_3 上，而 A 和 C 是在 t_2 和 t_4 上，B 和 C 在 t_2 和 t_3 上。因而，可以这样说，程序的并发执行是一组在逻辑上互相独立的程序，或程序段在执行过程中，其执行时间在客观上互相重叠，即一个程序段的执行尚未

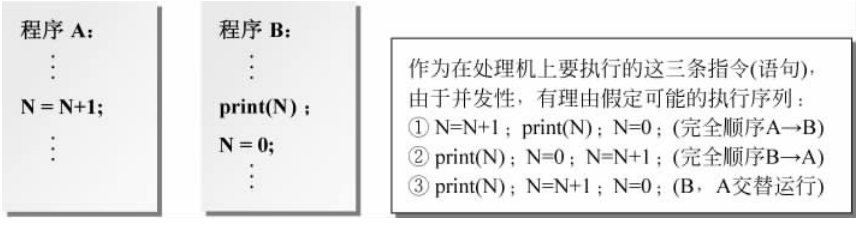
结束，另一个程序段的执行已经开始的这种执行方式。

虽然程序的并发执行带来了系统吞吐量的提高，但同时也产生了与顺序执行有所不同的新的特征（对比顺序执行的特征）。

（1）间断性。程序在并发执行时，由于它们共享资源或为完成某一项任务而合作，致使在并发程序之间存在相互制约的关系。如图 3.5 中，I、C、P 是 3 个相互合作的程序，当计算程序完成 C_{i-1} 的计算后，如果输入程序 I 尚未完成对 I_i 的处理，则计算程序无法进行 C_i 处理，致使计算程序暂停运行。同理，如果打印程序完成了 P_i 的打印后，若计算程序尚未完成对 C_{i+1} 的计算，打印程序就无法进行对 C_{i+1} 的计算结果的打印，运行后续的 P_{i+1}。一旦某程序的暂停因素被解除，即可引起后续程序的运行，这时就从程序的顺序性演变成间断性。对于程序的间断性，更一般的情形参见图 1.6。

（2）失去封闭性。程序在并发执行时，是多个程序共享系统中的各种资源，因而这些资源的状态将由多个程序来改变，致使程序的运行失去了封闭性。当处理机资源被其他程序占用时，有条件运行的任何程序都必须等待。

（3）不可再现性。程序在并发执行时，由于失去了封闭性，也导致失去了可再现性。



例 1　有两个程序 A 和 B，它们共享一个变量 N，并假定当前值为 n。其中 A，B 的操作分别如下：

由于并发性，程序 A 和 B 以不同的速度运行，因而产生了不同的结果。

① 3 条指令对于 N 的操作结果分别为 N＋1、N＋1、0，最终 N 的结果为 0。

② 3 条指令对于 N 的操作结果分别为 N、0、1，最终 N 的结果为 1。

③ 3 条指令对于 N 的操作结果分别为 N、N＋1、0，最终 N 的结果为 0。

上述情形说明，程序在并发执行时，由于失去了封闭性，其计算结果已与并发程序的执行速度有关，从而使程序也失去了可再现性。

例 2　假定一个航班售票系统运行在一个多终端分时系统上共享主机系统数据（库）资源，并在一个城市有两个终端售票机 B1、B2，任意航班的座位按顺序预订，简单座位形式如图 3.8 所示；其中黑色部分表示已售（值为 1），空白部分为未售（值为 0）。设定有 n 排，每排 m 个座位（$n,m\geqslant 1$），又假定座位按顺序预订。两个终端售票机 B1、B2 有下面虚线框的公共数据（库）与相同的预订程序。

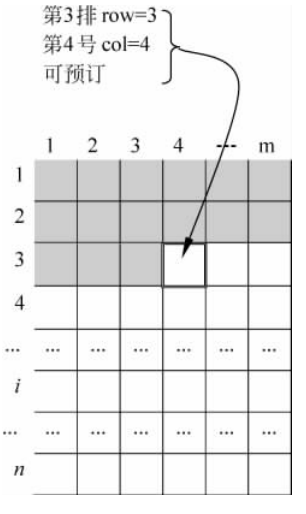


图 3.8　航班座位示意图

```
var
  row, col : integer;
  ticket[n][m] : integer;
        ...
  procedure booking
1: begin
2:    if row <= n
3:       begin
4:          ticket[row][col]:= 1; 赋值 1 表示已售
5:          write ("座位:" row"排", col "号");
6:          col = col mod m + 1;
7:          if col = 1;
8:             row= row + 1;
9:       end
10:   else
11:      write ("座位已售完!");
12: end
```

共享数据

在上面的程序中，mod 表示模运算。假定当前可预定的位置在第 3 排第 4 个位置，即 row＝3，col＝4，如图 3.8 所示。显然，如果 B1、B2 完全顺序执行，即 B1→B2，或 B2→B1，则两个终端各自为一个顾客打印一张不同座位的票，不会有任何问题。

但如果 B2 售票机正在执行该程序，执行完语句 5，意味第 3 排第 4 号已售给终端 B2 所在的顾客，这时假定由于时间片到等原因暂停，B1 终端机开始执行，由于 row 和 col 的值仍是 3 和 4，故 B1 在执行到语句 5 时，又将该座位售于终端 B1 所在的顾客，如下所示。

```
B2:  1: begin
     2: if row <= n
     3:    begin
     4:      ticket[row][col]:= 1;
     5:    write ("座位:"row"排", col "号");
```

```
B1:  1: begin
     2: if row <= n
     3:    begin
     4:      ticket[row][col]:= 1;
     5:    write ("座位:"row"排", col "号");
```

这就由于并发性产生了公共数据值的错误造成一张票售予了两个顾客。这种问题的解决将在第 6 章中详细讨论。

细心的读者会发现，对于不同座位分布情况，以及两个终端程序的不同语句交叉执行，可能还会产生其他错误。

该例子一再说明了如下问题：在某些情况下，程序的并发执行使得其执行结果不再具有封闭性和可再现性，且可能造成程序出现错误。例子中的程序段并发执行出现错误结果是由于两程序段共享数据(库)资源，从而使得执行结果受执行速度影响。一般情况下，并发执行的各程序段如果共享软、硬件资源，都会造成其执行结果受执行速度影响的局面。显然，这是程序设计人员不希望看到的。为了使得在并发执行时不出现错误结果，必须采取某些措施来制约、控制各并发程序段的执行速度。

5. 程序并发执行的条件

并发的特性可使程序失去了结果的可再现性，因此为保持可再现性，就需要考察程序并

发执行的条件，即仅当满足一定条件时，并发程序才可能保持可再现性。可以将并发执行过程描述为：

```
S0
    Cobegin
    P1; P2; …; Pn;
    Coend
Sn
```

这里，S_0，S_n 分别表示并发程序段 P_1，P_2，…，P_n 开始执行前和并发执行结束后的语句。P_1；P_2；…；P_n 也可以由同一程序段中的不同语句组成或表示。

1966 年 Bernstein 提出了两相邻语句 S_1、S_2 可以并发执行的条件：

将程序中任一语句 S_i 划分为两个变量的集合，$R(S_i)$和 $W(S_i)$分别为 S_i 的读集和写集。其中，

$R(S_i)=\{a_1,a_2,\cdots,a_m\}$是语句 S_i 在执行期间对其进行读的变量；$a_j(j=1,\cdots,m)$；

$W(S_i)=\{b_1,b_2,\cdots,b_n\}$是语句 S_i 在执行期间对其进行写的变量；$b_j(j=1,\cdots,n)$。

如果对于语句 S_1 和 S_2，有

① $R(S_1)\cap W(S_2)=\{\ \}$，即 S_1 需要的变量不是 S_2 修改的变量；

② $W(S_1)\cap R(S_2)=\{\ \}$，即 S_2 需要的变量不是 S_1 修改的变量；

③ $W(S_1)\cap W(S_2)=\{\ \}$，即双方都不修改相同的变量。

若有两条语句 $C=a-b$ 和 $W=c+1$，它们的“读集”和“写集”分别为：

```
R(C=a-b)={a,b}; W(C=a-b)={c}
R(w=c+1)={c}   ; W(w=c+1)={w}
R(C=a-b)∩W(w=c+1)={ }
R(w=c+1)∩W(C=a-b)={c}
```

所以两条语句不能并发执行。

如果并发执行的各程序段中语句或指令满足上述 Bernstein 的 3 个条件（同时成立），则认为并发执行不会对执行结果的封闭性和可再现性产生影响（证明略）。但在一般情况下，这个条件对于软件设计（模块化）过于苛刻，系统要判定并发执行的各程序段是否满足 Bernstein 条件是相当困难的。从而，如果并发执行的程序段不按照特定的规则和方法进行资源共享和竞争，则其执行结果将不可避免地失去封闭性和可再现性。

根据 Bernstein 条件，我们不能通过限制程序的编写方式和内容来获得系统的可再现性，因此解决此问题可能需要依靠控制程序的执行过程来解决。因为操作系统用户随机性与各道程序逻辑独立的特点将使得每个用户程序所使用的软硬件资源都受到其他并发程序的共享和竞争，从而得到非预料的或不正确的结果。为了控制和协调各程序执行过程中的软硬件资源的共享和竞争，显然，必须应有一个描述各程序段执行过程和共享资源的基本单位。

从上述讨论可以看出，由于程序的顺序性、静态性以及孤立性，用程序段作为描述其执行过程和共享资源的基本单位既增加操作系统设计和实现的复杂性，也无法反映操作系统所应该具有的程序段执行的并发性、用户随机性以及资源共享等特征。也就是说，用程序作为描述其执行过程以及共享资源的基本单位是不合适的。需要有一个能描述程序的执行过程且能用来共享资源的基本单位。这个基本单位被称为进程（或任务）。

3.2.2 进程的描述

1. 进程的定义

进程的概念是20世纪60年代初期，首先在MIT的Multics系统和IBM的TSS/360系统中引用的。从那以来，人们对进程下过许多各式各样的定义。

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

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

(3) 进程是一抽象实体，当它执行某个任务时，将要分配和释放各种资源(P. Denning)；

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

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

(6) 进程是程序在一个数据集合上的运行过程，是系统进行资源分配和调度的一个独立单位；

(7) 进程在给定的活动空间和初始状态下处理机的一次执行过程等。

以上进程的定义，尽管各有侧重，但在本质上是相同的，即主要注重进程是一个动态的执行过程这一概念。

进程和程序是两个既有联系又有区别的概念，主要区别如下。

(1) 进程是一个动态概念，而程序则是一个静态概念。程序是指令的有序集合，没有任何执行的含义。而进程则强调执行过程，它动态地被创建，并被调度执行后消亡。

(2) 进程具有并行特征，而程序没有。由进程的定义可知，进程具有并行特征的两个方面，即独立性和异步性。也就是说，在不考虑资源共享的情况下，各进程的执行是独立的，执行速度是异步的。显然，由于程序不反映执行过程，所以不具有并行特征。

(3) 进程是竞争计算机系统资源的基本单位，从而其并行性受到系统资源的制约。这里，制约就是对进程独立性和异步性的限制。

(4) 不同的进程可以执行多个独立的程序，也可以包含同一程序，只要该程序所对应的数据集不同。当一个程序多次被执行时，每次执行都可以是一个不同的进程。

(5) 作为被控制和管理的实体，除了程序本身和所需数据集之外，应包括控制和管理信息。

由此，进程的特征可以简单概括为动态性、并行性、独立性、异步性和结构性。

2. 作业和进程的关系

作业是用户需要计算机完成某项任务时要求计算机所作工作的集合。进程是已提交完毕程序的执行过程的描述，是资源分配的基本单位。区别与关系如下。

(1) 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业之后，系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体，是向系统申请分配资源的基本单位。任一进程，只要它被创建，总有相应的部分存在于内存中。

(2) 一个作业可由多个进程组成。且必须至少由一个进程组成，但反过来不成立。

(3) 作业的概念主要用在批处理系统中。而进程的概念则用在几乎所有的多道系统中。

3. 进程的类型

如果说操作系统是由众多个功能模块构成的大型程序的话，同用户程序一样，也需要在

处理机上运行(运行的单位也是进程),只是它们各自的目的不同。因此,可将进程分成系统进程和用户进程。

(1) 系统进程:系统进程属于操作系统的一部分,用于完成系统的某些功能。一个系统进程的任务一般都是相对独立的,且在生存期内一直保持该功能,因而有的通常对应一个无限循环程序。在现代操作系统中设置了很多系统进程,它们都运行在管态。由于系统进程是用于管理和维护系统的任务,因而它们的优先级一般高于用户进程。

(2) 用户进程:用户进程一般是用户从外存调入内存后,通过建立相应的信息结构成为系统资源分配和调度的运行实体。当然,所谓的用户进程,并不一定是指用户所编写的程序,例如,用户调用 C 语言的编译程序对其所编写的程序进行编译,而编译程序的运行是在用户态下运行的,因而也看成是用户进程。从系统层次上看,在操作系统之上运行的应用进程均称为用户进程。

3.3 进程控制块和状态转换

对于控制和管理的基本单位——进程,系统中需要有描述进程存在和能够反映其变化的物理实体,它由 3 部分组成:进程控制块 PCB(Process Control Block)、进程描述表(Process Descriptor)、有关程序段和该程序段对其进行操作的数据结构集。

(1) 进程控制块 PCB。进程控制块是操作系统用于控制和管理进程的一种数据结构,是进程存在的唯一标志。因此,只有操作系统可以访问,而用户是不感知,也是无法访问的。它属于系统空间。

(2) 程序代码。程序是进程执行的实体。在现代操作系统中,一般都要求程序地址是可以相对浮动的(从"0"相对编址),以便可以存放在系统空间的任何位置上。

(3) 数据集合。一般是进程私用的,包括局部变量、栈等数据结构。

进程可以有两种表示方法,如图 3.9(a)将代码和数据看成一个整体。图 3.9(b)则强调了代码部分的可共享性。需要说明的是,虽然进程是由以上三个部分组成,但要注意 PCB 是属于系统空间,而程序和数据是属于用户空间的。

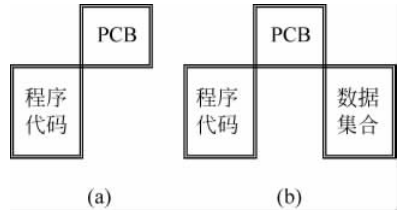


图 3.9 进程的组成

进程控制块包含了有关进程的描述信息、控制信息以及资源信息,是进程动态特征的集中反映。系统根据 PCB 感知进程的存在和通过 PCB 中所包含的各项变量的变化,掌握进程所处的状态以达到控制进程活动的目的。由于进程的 PCB 是系统感知进程的唯一实体,因此,在几乎所有的多道操作系统中,一个进程的 PCB 结构都是全部或部分常驻内存的。

3.3.1 进程控制块 PCB

PCB 包含进程的描述信息、控制信息及资源信息,有些系统中还有进程调度所使用的现场保护区。PCB 集中反映一个进程的动态特征。在进程并发执行时,由于资源共享,带来各进程之间的相互制约。显然,为了反映这些制约关系和资源共享关系,在创建一个进程

时，应首先创建其 PCB，然后才能根据 PCB 中信息对进程实施有效的管理和控制。当一个进程完成其功能之后，系统则释放 PCB，进程也随之消亡。

1. 进程控制块

对于不同的操作系统，PCB 中信息的数量和内容不尽相同。但一般来说，系统规模越大，功能越强，其 PCB 的信息数量也越多。图 3.10(a)列出了一般操作系统所具有的基本内容。

(1) 标识信息：包括进程标识符，它是唯一标识一个进程，也是在系统中存在的唯一标志。通常，进程标识符存在两个标识，一个是由用户(进程)访问该进程时使用的外部标识符，称为进程名，如输入进程、计算进程、打印进程等(参见图 3.10(b) Windows 系统进程名)。另一个是内部标识符，通常为一整数，称为进程号。所属用户也存在用户标识，同样有外部和内部用的用户标识。一个进程有唯一的一个用户号相对应，而一个用户号可以有多个进程号对应。家族联系是用于记录父进程与子进程的联系指针。

标识符信息	进程标识符	进程名
		进程号
	用户标识	用户名
		用户号
	家族联系	父进程
		子进程
处理机状态信息（现场）	通用寄存器	
	指令计数器	
	程序状态字	
	用户栈指针	
进程调度信息	进程状态	
	进程优先数（级/权）	
	等待原因	
	调度算法参数等	
进程控制信息	程序和数据地址	
	进程同步和通信机制	
	资源清单	
	链接指针	
	访问权限	
	打开的文件	

(a)

(b)

图 3.10 进程控制块信息

(2) 处理机状态信息：处理机状态信息主要是由处理机各种寄存器的内容所构成。处理机在运行时，许多信息都存放在寄存器中。当进程因等待某个事件而进入等待状态或因某种事件发生被中止在处理机上的执行时，为了以后该进程能在被打断处恢复执行，需要保护当前进程的 CPU 现场(或称进程上下文)。包括程序计数器 PC、程序状态字 PSW 等。

(3) 进程调度信息：进程调度信息包括进程的状态、进程的优先级以及进程等待的时间和运行的时间等，提供进行进程切换的依据。

(4) 进程控制信息：进程控制信息包括程序和数据在内存或外存上的地址，以及占用的空间大小，具体内容与存储管理有关。进程同步和通信机制，用于实现进程间通信，如消息队列、信号量等。资源清单，用于记录除 CPU 之外，在其生存期间所使用的系统资源，以

及使用时间等，一般用于记账。链接指针，用于构建 PCB 队列，它给出了下一个进程 PCB 的首地址。访问权限，用于控制对系统资源的访问权力，如对文件的访问属性，读、写、执行等，这与系统保护机制有关。打开的文件，用于记载当前进程使用的文件，通过它与内存管理表项建立联系，通过该表可以找到文件在外存上的地址。

由于进程控制块 PCB 包含了系统所需要的关于进程的所有信息，因此，进程控制块是操作系统最重要的数据结构。实际上，操作系统中的许多模块，包括那些涉及调度、资源分配、中断处理、性能监控和分析模块，都可能访问 PCB 中的信息，所以系统需要考虑对 PCB 加以保护以防止被错误的（系统）例程改写（用户程序是肯定禁止访问的），这在设计操作系统时是需要认真考虑的。

如前所述，进程控制块 PCB 是系统感知进程存在的唯一实体。通过对 PCB 的操作，系统为有关进程分配资源从而使得有关进程得以被调度执行。当进程执行结束后，则通过释放 PCB 来释放进程所占有的各种资源。

2. 进程控制块的组织

为实现对进程的管理，系统需要对进程控制块以某种策略进行组织。通常，组织方式主要有以下两种。

（1）链接方式。在操作系统中，通常将具有相同状态进程的 PCB 根据 PCB 结构内部的链接指针，将这些进程的 PCB 链接在一起，这样就形成了就绪队列、阻塞队列以及空闲队列。由于不同操作系统的规模和功能不同，对于就绪和阻塞队列的建立，可能由于其原因不同，可能进一步建立若干个就绪队列和阻塞队列。对于就绪队列，可以根据其优先级/权的不同而分别建立若干个就绪队列。同理，对于阻塞队列，可根据被阻塞的原因，如等待 I/O、资源使用互斥等待等，将具有相同阻塞原因的进程的 PCB 组织在一起。

图 3.11 给出了链接形式的 PCB 组织方式。队列的第一 个 PCB 由一个指示字（指针）来指向，队列的尾用一个 0，或 Λ 符号来表示。队列可按先进先出（First-in First-out，FIFO）

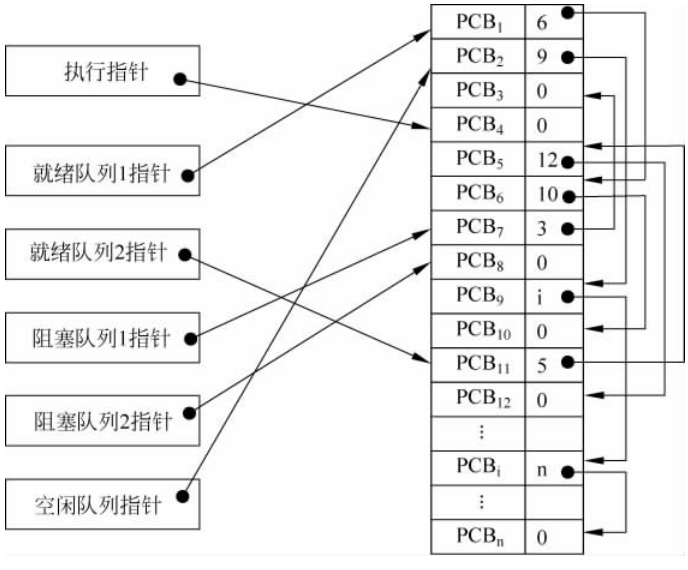


图 3.11　链接队列的 PCB 示意图

等策略构建。应当说明的是，虽然这里使用了“队列”这一术语，但 PCB 入队列和出队列不一定完全按照 FIFO 的次序，这与操作系统对进程的管理策略有关，不失一般性，这里仍采用队列这一术语。

(2) 索引方式。索引方式就是根据进程的不同状态建立几个索引表，如就绪索引表、阻塞索引表等。索引表中的每一项指示一个 PCB 在内存的首地址，而各索引表的起始地址存放在内存的专用指示单元中。图 3.12 给出了索引方式的 PCB 组织。

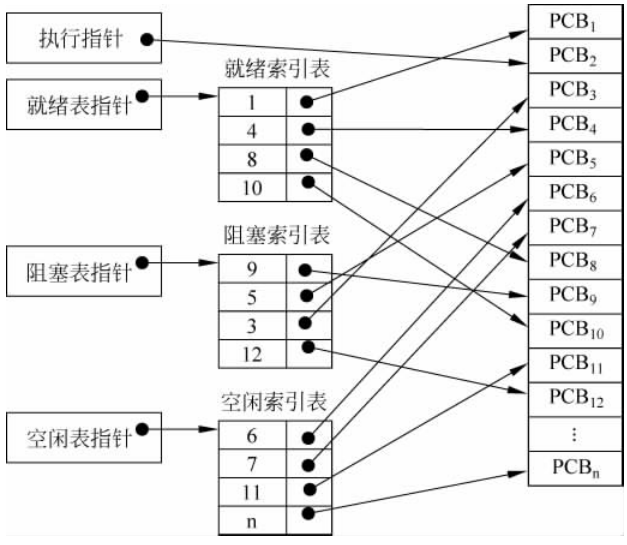


图 3.12 索引方式的 PCB 示意图

3.3.2 进程状态及其转换

1. 进程状态

尽管每个进程是个独立的实体，有其自己的程序计数器、运行空间等，但正如前所述，进程或进程之间可能由于等待 I/O 操作、竞争资源，以及相互协作等原因产生了“走走停停”的动态性。因此，一个进程在生存期内可有多种状态，这些状态刻画了进程在生存期内整个运行轨迹。进程的状态转换(变迁)是操作系统根据进程 PCB 结构中的状态值控制进程。进程在生命存期内至少具有 3 种基本状态：执行状态、阻塞状态和就绪状态。图 3.13 给出了进程 3 个基本状态的状态图。

(1) 运行状态。运行状态是系统内一个进程分派(即为调度)到了处理机(CPU)正在运行的状态。在单处理机系统内，显然，只能有一个进程处于“运行”状态(在 PCB 表中的状态项为“运行”)；而在多处理机系统内，可有多个进程处于“运行”状态，但运行状态进程的个数一般少于处理机的个数。

(2) 就绪状态。就绪状态是指进程可运行，只是处理机被其他进程占用。处于这种状态的进程，

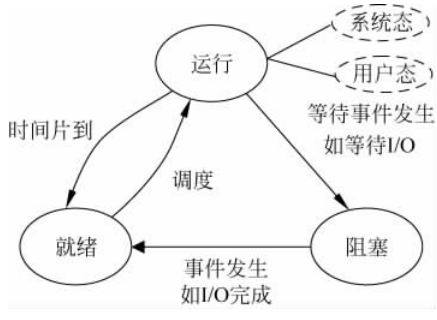


图 3.13 进程三状态及转换图

其他的运行条件都已满足(所需的其他资源都已具备,如运行空间、所需文件、信息以及外部设备等),正在等待分派处理机(资源)。处于就绪状态的进程可能很多(一个系统中,根据硬件资源,尤其是内存,可设定多达近百个进程),因此需要根据 PCB 的组织方式将其排成一个,或多个就绪队列(见图 3.11,图 3.12)。

(3) 阻塞状态,也称等待/封锁/睡眠状态(当然,不同系统的细划分不同,可能有所差别),是指进程因等待系统内某(些)事件的发生而暂时无法运行。这与处于就绪状态的进程不同,被阻塞的进程由于不具备运行条件,即使处理机空闲也无法在处理机上运行。如等待输入数据文件,当前没有数据就无法在处理机上进行相应的数据处理。同样,处于这种状态的进程也可能很多,也同样需要根据系统对 PCB 的组织方式,由于原因不同而将其排成一个,或多个阻塞队列。

运行、就绪、阻塞是进程的 3 个基本状态,而操作系统的规模和设计目的不同,进程状态的设置可能多于 3 个。在有些系统中,还增加了两个基本状态,新建状态、完成状态/或终止状态/结束状态,这两个状态对进程管理是非常有用的。

(4) 新建状态对应于刚刚定义的进程,还未进入(准备进入)就绪队列的状态,此时系统还未完全将其进程的全部工作完成(如内存空间尚待分配等)。例如,如果一个新用户试图登录到分时系统中,或一个新的批处理作业被选中准备投入内存,则操作系统可以首先执行一些必要的辅助工作,将标识号关联到进程,分配和创建管理进程所需要的所有表。此时,进程处于新建状态。只有当进程的代码和数据进入内存后,才能进入就绪队列,成为在处理机上运行的候选者。新建进程的常见事件见表 3.1。

表 3.1 导致进程创建的原因

事　　件	说　　明
新的批处理作业	通常位于磁带,更一般地位于磁盘上的批处理作业流提供给操作系统。当操作系统准备接纳新任务时,将调入选中的若干作业
交互登录	终端用户登录到系统
操作系统因提供一项服务而创建	操作系统可以创建一个进程,代表用户程序执行一个功能(输出进程帮助用户实现数据输出,使用户无须等待)
由现有的进程派生	基于模块化,或为了并发性,用户程序可以指示创建多个进程

(5) 完成状态。完成状态是指进程正常,或非正常结束而终止的状态。处于该状态的进程还未从系统中消失,可能由于一些善后工作尚未完成(如记账,未完成的输出等)。但处于这种状态的进程以后不再被调度执行。表 3.2 给出了进程完成,或终止的一些原因。

表 3.2 导致进程终止的原因

事　　件	说　　明
正常完成	进程执行完任务,自行执行一个操作系统服务调用,表示已经结束运行
无可用内存	系统无法满足进程所需要的内存空间
父进程终止	当一个父进程终止时,操作系统自动终止所有子孙进程
父进程请求	父进程具有终止后代进程的权利
时间超出	进程等待某一事件发生的时间超过了规定的最大值
地址越界	进程试图访问不允许访问的内存单元

续表

事　　件	说　　明
保护权限错	进程试图使用不允许使用的资源或文件，或以一种不适当方式使用，如向只读文件进行写的操作
数据溢出	执行了除“0”，或机器硬件无法表示的数据
I/O 失败	在输入输出期间发生错误，如查不到所需求的文件，I/O 设备经过多次启动失败（一般 3～5 次），从打印设备读取数据等
特权指令/无效指令	进程在用户态执行特权指令，或执行了一条不存在的指令（如进入数据区，执行数据）
数据误用	进程使用未初始化，或类型错误的数据
系统操作员	由于某些原因，系统操作员，或操作系统终止进程（如系统可能存在死锁）

图 3.14 给出了 5 个状态的进程状态图。

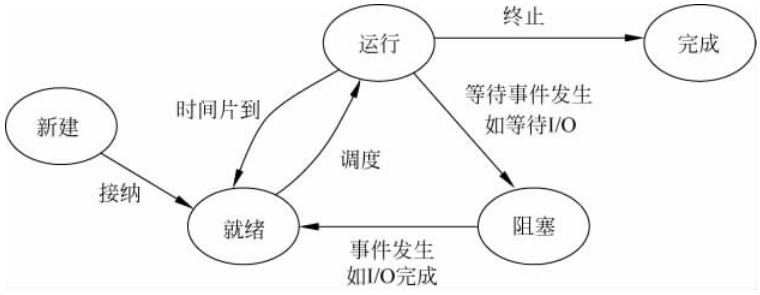


图 3.14　进程 5 状态及转换图

一个操作系统设置多少个状态，是与系统对进程的管理方式有关，也与系统资源的利用有关，但有一点是明确的，系统中设置过多的状态会造成系统参数和状态转换过程的增加。

2. 进程状态转换

进程在生存期间，可以多次地从一个状态转换到另一个状态，即多次地处于运行状态、就绪状态、阻塞状态，反映了并发程序走走停停的运行轨迹。进程不断地从一个状态转换到另一个状态是有条件或原因的。这些状态随着进程的执行和外界条件发生变化而转换。事实上，进程的状态转换是一个非常复杂的过程。从一个状态到另一个状态的转换除了要使用不同的控制过程，有时还要借助于硬件才能完成。

（1）空→新建。通常有 4 个事件可能导致创建一个新进程，见表 3.1。

（2）新建→就绪。当操作系统确定系统当前由于内存等资源的条件可以满足，可以接纳一个进程时，就把一个进程的新建状态修改为就绪状态。一般系统对进程的数目有一个限定，以防止由于进程过多而使得系统资源过分紧张，系统效率下降。

（3）就绪→运行。选择一个在就绪队列中的进程，分派处理机给它，同时将就绪状态修改为运行状态。处于运行状态的进程也称为当前进程。至于如何从就绪队列中的进程选择一个进程投入运行属于调度策略问题，这个问题将在第 4 章加以讨论。

（4）运行→就绪。这类状态转换最通常的原因就是，正在运行的进程到达了系统限定的一个最长时间——时间片，该进程就被强制剥夺在处理机上的运行的权力，从运行状态进入就绪状态，即将运行状态修改为就绪状态。实际上，几乎所有的多道程序操作系统都实行了这类时间限定。这里要说明的是，从运行到就绪状态的转换，还可能由于其他原因，例如

根据进程的优先级，如果当前就绪队列由于系统的某个事件而出现了一个进程，它的优先级高于当前进程优先级，是否立即剥夺当前进程。这并不是所有操作系统都需要考虑的，仍属于系统调度策略问题。

另外要说明的是，运行到就绪状态的转换，仅是当前进程的时间到，而不是由于其他原因。换句话说，该进程是处于“可运行”状态。

(5) 运行→阻塞。如果当前进程由于需要等待它所请求的事件发生，或因为某(些)条件未能得到满足，则进入阻塞状态。由运行进入阻塞状态的原因可能很多，如请求操作系统的一个服务，而操作系统无法立即给予服务；也可能请求了一个无法立刻得到的资源，如文件，或一块内存空间；或由于等待进程间通信的信息等，都可能被阻塞。

(6) 运行→完成。正在运行的进程由于任务执行完成，或由于其他原因无法继续在系统中运行下去，系统就将当前进程从运行状态转变为完成状态。处于完成状态的进程并不立刻从系统退出而消失，因而存在一个完成状态。进程结束的一些常见的原因见表 3.2。

(7) 阻塞→就绪。当处在阻塞队列中的进程因所等待的事件发生时，系统根据事件的原因查找阻塞队列中的进程，并将其由阻塞状态转换为就绪状态，进入就绪队列参与竞争处理机(资源)。

进程在系统中的并发运行就是在状态转换过程中不断地向前推进，直至完成到退出系统。表 3.3 给出了进程状态之间转换的简要描述。

表 3.3　进程状态转换简要描述

状 态 转 换	事　　件	动　　作
空→新建状态	创建一个新进程	建立系统管理数据结构
新建状态→就绪状态	接纳一个新进程	进入就绪队列
就绪状态→运行状态	进程调度	分派处理机
运行状态→就绪状态	时间片到时，出现高优先级进程	系统剥夺处理机
运行状态→阻塞状态	等待 I/O 等事件	放弃处理机，进入阻塞队列
阻塞状态→就绪状态	等待 I/O 事件发生	进入就绪队列
运行状态→完成状态	进程任务完成，或非正常终止	释放资源

3. 进程的挂起状态

前面描述的 3 个基本状态(图 3.13)提供了一种构造进程活动和模型的系统方法，并指导操作系统设计与实现。许多操作系统都是按照这 3 种状态进行具体设计和构造的。但是，仅通过这种模型建立的系统尚不充分。一方面，从系统的角度，系统的处理机、内存等系统硬件资源的利用率得不到充分发挥，甚至出现闲置状态，使系统效率降低。

另一方面，处在活动空间的进程可能由于某(些)原因暂时静止下来，不处于活动空间中，但也不是从系统中彻底退出，这就导致了对 3 种状态模型的扩充，引入了挂起状态。

引入挂起状态的目的就是使一些进程已占用的系统资源让出部分，或全部(PCB 仍在系统中)，以供其他进程利用让出的系统资源，提高系统的整体效率。

为了说明增加挂起状态的合理性，考虑一个简单的存储管理模式；多道程序进入活动空间时，程序代码一次全部装入内存，这样，多个进程在运行期间，由于竞争 I/O 操作，有理由假定，系统在某些时刻，所有进程都在等待 I/O。而处理机在空闲。这是因为，处理机的

速度要远远地高于 I/O 的速度，致使所有进程的“计算”都已完成，都在等待 I/O 而处于“阻塞”状态。尽管系统的 3 状态模型可以使进程进入阻塞队列而处于活动空间，但处理机在空闲。又在后备空间(外存)上希望“计算”的进程又由于系统此时无法再容纳新的作业进入内存而无法得到运行。这种情形在系统中是常见的，因此，即使在多道情况下，大多数时间处理机仍处在空闲状态。

解决这个问题的方法有两种。

(1) 通过硬件方法扩充主存，装入更多的进程。但这种方法有两个缺陷；首先是价格问题，更重要的是，更大的内存将随着程序对内存空间更大的要求而导致装入更大的进程，而不是装入更多的进程。

(2) 通过软件的方法——交换，即将主存中某(些)尚不在运行的进程，特别是一些处于阻塞状态的进程，将其部分，或全部转移到交换区(如磁盘辅助空间)，这就产生了进程暂时挂起的状态。

图 3.15 给出了具有挂起状态的进程状态图。

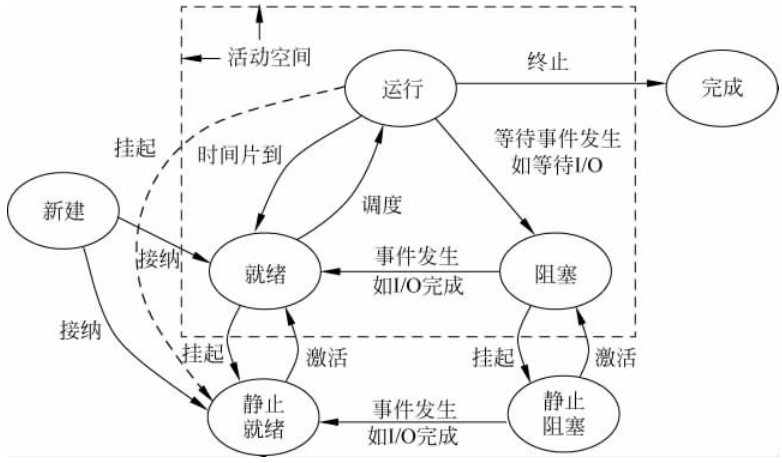


图 3.15 具有挂起状态的进程状态转换图

在引入挂起状态后，同时产生了从活动到静止，或相反过程的状态转换。

(1) 活动阻塞→静止阻塞。如果当前系统中没有就绪态进程，就将处于阻塞态的进程至少挂起一个，而进入静止阻塞状态，为没有被阻塞的进程让出主存空间。如果操作系统确定当前正在运行的进程，或就绪态进程为了维护基本的性能要求而需要更大的内存空间，则即使存在可运行状态的就绪进程也可能出现这样的状态转换而被挂起进入静止阻塞。

(2) 静止阻塞→静止就绪。同基本状态转换一样，如果等待的事件发生了，则将处于静止阻塞的进程修改为静止就绪状态。

(3) 静止就绪→活动就绪。如果主存中没有就绪进程，一般地，操作系统需要调入一个进程。而当处于静止就绪状态的进程的优先级高于(活动)就绪进程的优先级时，操作系统则往往将处于静止就绪的进程通过激活而将其转换为(活动)就绪状态。

(4) 活动就绪→静止就绪。通常，操作系统更倾向于挂起阻塞态进程而不是就绪态进程，因为就绪态进程属于“可运行”状态，而阻塞态进程占据主存空间而又处于暂时“无法运行”状态。但有两种情况需要这种转换；一是得到主存更大空间的唯一方法是挂起一个就

绪进程；二是如果能够确定处于高优先级阻塞状态的进程可以很快进入就绪状态，则可能会挂起处于低优先级的就绪进程，而不是挂起高优先级阻塞态进程。

上面的几种转换是常见的，还需要考虑的其他转换。

(5) 新建→活动就绪/静止就绪。当创建一个新进程时，该进程可以进入(活动)就绪队列，也可以进入静止就绪队列。系统初始执行期间，操作系统更倾向于建立更多的就绪进程来维护大量的未被阻塞的进程。这样的策略会使以后新的进程由于主存空间不足而无法进入，这时就使用了新建→静止就绪转换。

(6) 静止阻塞→活动阻塞。这种情况较少发生。如果一个进程处于阻塞，又不在主存(静止阻塞态)，调入它进入主存似乎意义不大。但有下面的情况是需要考虑的，即当前进程终止，释放了主存空间，如果静止阻塞队列中存在一个进程，它的优先级比静止就绪队列中的任何一个进程的优先级更高，且操作系统确定等待的事件很快可以发生，则将静止阻塞的进程而不是静止就绪的进程调入主存是合理的。

(7) 运行→静止就绪。通常，当一个运行的进程运行到所规定的时间段时，进入就绪队列。但如同上面考虑的情形，如果在静止阻塞队列中有优先级更高而不会再被阻塞，则操作系统会立刻抢占这个进程，而让当前运行到期的进程进入静止就绪，以便释放主存空间。

(8) 各种状态→完成。在正常情况下，一个运行的进程正常，或非正常结束，都将进入完成状态。但正如表 3.2 中所列出的进程终止事件，如果父进程终止，或被创建它的进程终止，则表明一个进程可以在任何状态下终止而进入完成状态。

总之，引入挂起的一个主要原因是提供更多的主存空间，以调入可运行的进程，或为其他进程分配更多的主存空间。表 3.4 列出了进程挂起的一些原因。

表 3.4 挂起的原因

事　　件	动　　作
对换	操作系统要求释放更多的主存空间，以调入并执行处于就绪态的进程
其他操作系统的原因	操作系统有时需要挂起某(些)进程，检查运行中资源的使用情况及记账，以便改善系统的性能，或者挂起被怀疑导致问题的进程
交互式用户要求	用户可能希望挂起一个程序的执行，目的是为了调试等
父进程的请求	父进程有时需要考察和修改子进程，或当要协调各子进程活动的活动时，需要挂起自己的子进程
定时	一个进程可能会周期地执行(如系统监视进程)，可能在等待下一个时间间隔时被挂起；或实时系统中负荷调节的需要，这时可由系统将一些不十分紧迫的进程挂起，缓解系统的负荷以保证系统实时任务的控制

3.4 进程控制

3.4.1 操作系统控制结构

操作系统控制计算机系统内部的事件，为处理机执行而进行进程调度，给进程分配资源(除处理机之外的其他资源)，并响应用户程序的基本服务请求。因此，操作系统是一个系统资源管理者，或管理实体。

1. 控制结构

操作系统为了管理进程和资源，必须掌握每个进程和资源当前状态的信息。通常采用的方法是，操作系统构造并维护所管理的每个实体的信息表。图 3.16 给出了相应概念和框架。

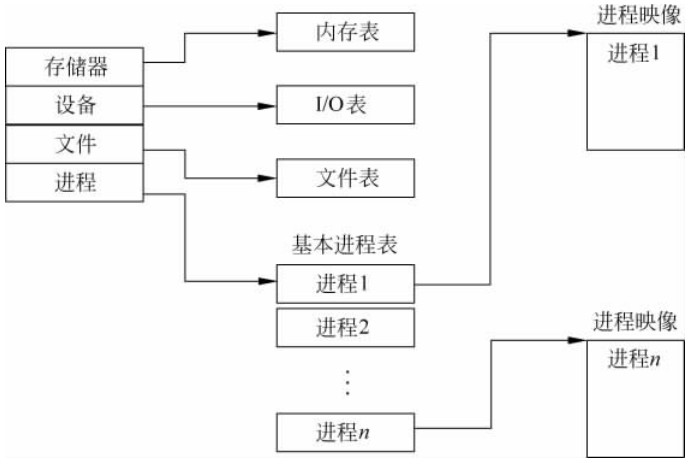


图 3.16 操作系统控制表的通用结构

操作系统维护着 4 种不同类型的表：内存、I/O、文件和进程。尽管不同的操作系统的实现细节上有所不同，但所有操作系统所维护的信息基本上是这 4 类。

(1) 内存表用于跟踪或记录主存和辅存。主存的一部分空间为操作系统保留，其余部分是进程可以使用的空间。内存表包括以下信息：分配给进程的主存、辅存空间大小、可以访问的共享内存区域以及存储管理的所有信息。

(2) 操作系统利用 I/O 表来管理系统内的 I/O 设备和通道。操作系统需要知道哪些 I/O 设备可用、哪些设备分配给了哪些进程。如果有 I/O 设备正在进行 I/O 操作，操作系统需要知道设备的使用状态、数据的源和目的的内存地址。

(3) 文件表记录了进程所拥有的文件的属性，包括文件在辅存中的位置、当前状态等。

(4) 进程表用于维护和管理系统中活动的各个进程，进程必须以某种组织形式链接起来，以便于定位和查找。如图 3.11 和图 3.12 所示。它也是操作系统用于控制和管理进程的主要依据和结构。应当说明的是，在操作系统管理中，这 4 种表是可以相互交叉引用的。

进程映像是程序、数据、栈和进程控制块的集合(也可称为进程上下文)。进程映像的位置依赖于使用的存储管理方法。最简单的情况，进程映像保存在邻近的或连续的存储块中，或保存在辅存中。对于操作系统控制和管理的对象，至少进程映像的一部分要保存在主存中。表 3.5 给出了进程映像组成及简要描述。

表 3.5 进程映像

元 素	描 述
用户数据	用户空间中的可修改部分，可能包括程序数据、用户栈
用户程序	将被执行的代码序列
系统栈	每一个进程有一个，或多个系统栈，用于保存参数、过程调用地址和系统调用地址
进程控制块 PCB	操作系统控制进程所需的数据和信息，见图 3.10(a)

2. 处理机的执行模式

从计算机系统安全和保护的角度出发，在进行计算机体系结构设计时，处理机的执行模式一般分为两种，内核模式（或核心态、系统态、管态）和用户模式（用户态、算态、目态）。两种模式的区别参见 1.5 节相关内容。显然，处理机在运行期间需要在内核模式和用户模式之间进行切换。

这里需要注意的是，进程切换（通过调度实现）与处理机执行模式的切换是不同的概念。在多道系统中，进程切换的时机通常在事件中断时。无论是何种事件的中断，系统都将进入内核模式，执行系统的中断处理。至于进程是否在中断发生时进行切换，与系统的调度方式和策略有关。如果一个用户进程在分时系统下运行，由于 I/O 事件被打断，处理机进入内核模式进行中断处理，然后，如果是该进程的时间片未到，也非抢占调度方式，则该进程可恢复断点继续在用户模式下运行，因此，进程切换与处理机模式切换是不同的。

使用两种模式的原因是很显然的，它可以保护操作系统及相关的各种数据表（见图 3.16）不受到用户进程的干涉，这也是操作系统管理和控制计算机系统的重要技术手段。

3. 操作系统内核

内核是操作系统的控制和协调中心，由它组织、启动和协调系统中各种活动。因此，现代操作系统广泛采用层次结构，将操作系统分为若干层次对应不同工作或功能。一般将与硬件紧密相关的模块安排在紧靠硬件的软件层次中，如中断处理程序、进程控制与调度程序、I/O 驱动程序等。通常将这一部分称为操作系统内核。图 3.17 是一个简单的层次图示。操作系统一些典型的内核功能见表 3.6。内核部分常驻内存，以提高系统运行效率，并对其加以特殊的保护。

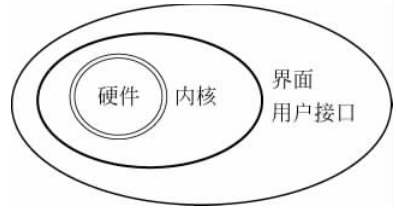


图 3.17　操作系统层次结构

表 3.6　操作系统内核典型功能

功　能	描　述
进程管理	进程派生和调度
	进程的创建和终止
	进程同步以及对进程间通信的支持
	PCB 的管理
内存管理	进程地址空间的分配、回收
	交换
	页、段的管理
I/O 管理	设备驱动
	缓冲区管理
	进程 I/O 设备、控制器、通道的分配、回收
支持功能	中断处理
	时钟管理（其中包括中断）
	监视

应当说明的是，操作系统内核分为强内核和微内核，或弱内核。微内核的基本构造是只有最基本的操作系统功能放在内核中。非基本的服务和应用程序在微内核之上，并在用户

模式下执行。尽管什么应该放在微内核，什么应该在微内核外，不同的设计有不同的分界线，但总的原则是传统上属于操作系统一部分的功能都是外部子系统，包括文件系统、I/O管理、存储管理和进程管理等大部分功能，只有其中一小部分放在内核中。

内核中有些功能由原语操作实现。原语本身也是由若干条指令构成，用于完成一定功能的一个过程。它与一般过程/函数的区别在于是原子操作。所谓原子操作就是不可分割的操作，要么不做，要么全做。

原语可分为两类：一类是机器指令级的，其特点是执行期间不允许中断，在操作系统中，它是一个不可分割的基本单位；另一类是功能级的，其特点是作为原语的程序段不允许并发执行。这两类原语都在系统态下执行，且都是为了完成某个系统管理所需要的功能和被高层软件所调用。

利用原语这一特征，通常把进程控制用程序段做成原语。用于进程控制的原语有创建原语、撤销原语、阻塞原语、唤醒原语、进程同步原语等。

在此应当强调的是，中断处理是操作系统内核中一项非常重要的功能，是系统中进程状态转换和进程切换的基础和核心，也是操作系统的“灵魂”。

3.4.2 进程创建与终止

1. 进程创建

1）进程图

在有的操作系统中。为了提高系统的并发性和系统整体效率，采用了进程运行过程中可以根据条件和需要动态地创建进程的方法。即进程可以创建子进程，子进程创建子进程，便形成了树状的进程家族结构——进程图。图 3.18 给出了进程图的示意图。

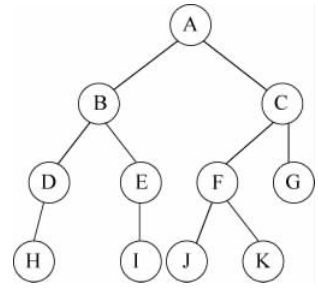


图 3.18 进程树

进程图是有向树，图 3.18 中的结点表示进程。如果进程 P_i 创建了进程 P_j，则称 P_i 为父进程，P_j 为子进程，它们之间画一条从 P_i 到 P_j 的有向边。树的根为进程家族的祖先。应当说明的是，进程树本身看起来似乎是简单的，与数据结构中树的结构相似，但这种关系隐含着可以继承父进程所拥有的资源，如内存空间、打开的文件等。父进程也可以动态地撤销（终止）子进程。在子进程终止时，归还从父进程得到的资源。另外，在父进程终止时，将同时撤销所有的子孙进程。为了标记进程这种家族关系，在进程控制表 PCB 中设有进程家族表项，指示其父进程和子进程。

并不是所有的操作系统都设有进程树结构，UNIX 是典型的带有进程树层次结构的系统，而 Windows 中不存在任何进程树结构的概念，所有的进程都是地位相同的。创建进程的权利（令牌或句柄）可以转移给其他进程，这样也就不存在进程层次结构了。

2）进程创建

关于引起进程创建的事件，不同的操作系统有细节上的不同，但基本上可以参考表 3.1 中的事件。

一旦操作系统发现有创建新进程的事件，便通过系统调用，或创建原语执行下面的一些步骤来实现新进程的创建。

(1) 给新进程分配(或产生)一个唯一的进程标识号 PID,从系统的空闲的 PCB 结构中摘出一个 PCB 表。

(2) 为新进程分配所需的资源;这里主要是为新建的进程分配主存空间,包括程序、数据和工作区。这时操作系统必须知道所需的内存空间的大小。一般由请求创建者提出,但对于交互性作业,用户可以不提出内存空间要求,由系统统一分配一定的空间。

(3) 初始化 PCB,包括进程名、用户名、父进程标识符、进程优先级、处理机状态(就绪)、程序执行地址、栈指针等。

(4) 将新进程插入就绪队列。

一个进程派生新进程后,有两种可能的执行方式:

① 创建者(父进程)和被创建者(子进程)并发执行;

② 创建者(父进程)等待它的某个或全部子进程终止。

2. 进程终止

导致进程终止的事件参见表 3.2。一旦系统发生进程终止事件,操作系统便通过终止进程的系统调用,或进程终止原语,执行下列步骤终止进程。

(1) 根据被终止进程的标识号,从系统的 PCB 表中检索到该进程的 PCB,若该进程正在运行,则立即终止它的运行。

(2) 回收该进程的全部资源(包括程序空间、外部设备等)。

(3) 若它还有子孙进程,则同时终止子孙进程,回收其全部资源。

(4) 将该进程的 PCB 从所在的结构中移出(暂存在系统其他空间),由其他进程搜集信息和进行善后工作(包括系统记账,未完的输出服务等),进入完成状态。待系统将该进程有关的一切工作完成后,收回该进程的 PCB,插入到空闲的 PCB 队列中,该进程就此消亡。

由于 UNIX 是具有进程层次结构(进程图),考虑 UNIX 进程终止的简单情况;对于任何一个用户进程,在完成相应任务后,在程序的尾部通过系统调用 exit(status)通知系统终止自己,其中 status 称为终止码,它是该终止进程向父进程传送的参数。父进程是由系统调用 wait 等待其子进程的终止。wait 系统调用返回被终止子进程的标识号(PID),所以父进程可以告诉系统是哪个子进程终止了。

3.4.3 进程的阻塞与唤醒

进程的创建和撤销实现了进程从无到有,从存在到消亡的生命周期。被创建后的进程最初处于就绪状态,此后有机会进入执行状态,是否能一直运行下去要受到系统内其他事件的影响。正如前所述,操作系统的并发特征反映在进程上就是“走走停停”。停的原因可能有很多种,其中的一个方面就是因为等待某个事件发生而进入了阻塞状态,同样是某个事件发生而被唤醒继续“走”。

1. 进程阻塞

引起进程阻塞和唤醒的事件通常有以下几种。

(1) 启动 I/O 操作。进程在运行过程中总是需要进行 I/O 操作的(至少要有某种形式的输出结果),如果进程只有在 I/O 完成之后,才能继续运行,则进程在 I/O 启动后,便自动进入阻塞状态而进入阻塞队列。等待 I/O 的完成。应当说明的是,进程并非直接启动 I/O 设备,而是在表明一个 I/O 过程,真正启动 I/O 设备是由系统执行的。

(2) 请求系统资源。进程在运行的过程中,向操作系统申请所需的资源,并要求为其服务。如果由于某(些)原因,系统不能立刻满足该进程,则就将该进程转换为阻塞状态。同样,如果进程希望启动 I/O 操作(与(1)相联系,却是两个过程),但所需的系统资源——I/O 设备无法满足(可能由于其他进程在独占使用);或进程所提出的内存需求系统暂时无法满足等原因,都将使当前运行的进程被阻塞,进入相关的阻塞队列。

(3) 同步约束。对于一些合作,或协作的进程之间由于推进的速度存在着差距,而使得有的进程必须等待而进入阻塞。如一"计算"进程需要另一个输入进程的输入数据进行"数据加工",如果输入进程的数据输入慢于计算进程的计算速度,则计算进程只有等待。

(4) 服务进程无服务。操作系统设置了一些特定功能的系统服务进程,每当服务进程完成服务任务(如发送数据)后,就将自己阻塞起来,进入阻塞队列等待新的任务的到来。

当前运行的进程在出现以上的事件时,就通过调用阻塞原语将自己阻塞。进程阻塞的过程如下:

(1) 立即停止当前进程的运行,保护 CPU 现场到该进程的 PCB 现场保护区;

(2) 将进程执行状态修改为阻塞状态,插入到相应阻塞队列中;

(3) 转进程调度程序重新选择一个就绪状态的进程投入运行。

2. 进程唤醒

唤醒进程的事件与阻塞相对应。当一个在阻塞队列中的进程,在所等待的事件发生后,由另外一个相关的进程通过调用唤醒原语将其唤醒。显然,进程的唤醒不能自己唤醒,是由其他相关进程唤醒的。

唤醒进程有两种方法:一种是由系统进程唤醒;另一种是由事件发生进程唤醒。当由系统进程唤醒等待进程时,系统进程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。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。唤醒原语既可以返回原调用程序,也可以转向进程调度,选择一个合适的进程。唤醒的过程如下:

(1) 将相关的阻塞进程从阻塞队列摘出;

(2) 修改阻塞状态为就绪状态,并插入到就绪队列中;

(3) 调度程序进行新的调度,或继续恢复原当前进程。

这里要说明的是,阻塞与唤醒应当成对出现。进程通过阻塞原语阻塞自己,而由其他相关进程唤醒,这在进行并发程序设计时是尤为需要注意的。

3.4.4 进程的挂起与激活

进程挂起的事件见表 3.4。当出现了引起进程挂起的事件时,系统利用挂起原语将所需要挂起的进程挂起。

1. 进程挂起

进程挂起的过程如下:

(1) 找到需要挂起进程的 PCB,将其从相应队列摘出;

(2) 将其空间归还系统(由系统存储管理模块负责回收);

(3) 判断被挂起进程的状态,如为阻塞状态,将其状态修改为静止阻塞;如为就绪,修

改为静止就绪；

(4) 申请交换区(外存)空间，将部分，或全部进程映像写到交换区，并将交换区地址记入 PCB 中；

(5) 如被挂起的进程为当前进程，则转向进程调度。

2. 进程激活

当出现激活事件时，若进程映像驻留在交换区，内存空间满足时，系统利用激活原语将指定的进程激活。激活的过程与挂起的过程基本是个对应相反的过程。

3.4.5 进程间的相互关系

在多道环境下存在多个并发进程，这些并发进程之间的关系可以从两个方面考察，结构(物理上)的相关性和逻辑的相关性。

1. 结构相关进程

1) 相关进程

这些进程之间存在着结构上的相关，通常为了充分发挥系统的效能，或系统资源的利用率，允许一个进程通过进程创建产生多个子进程，构成进程家族，例如图 3.18 所示。A 进程创建两个子进程 B 和 C，A 是父进程，B 和 C 是子进程；C 进程又创建两个子进程 F 和 G，它们都是 A 的子孙进程。显然，这些家族的进程是结构上相关进程，因而在这些进程之间才可能存在与其他进程不同的两个方面的特征：一是资源继承性；二是“消失”的同时性。需要说明的是，家族结构上的进程是可以与层次无关地并发运行。

2) 无关进程

不存在结构上联系的进程是无关进程。例如，两个不同的用户所建立的作业(进程)，它们各自在解决自己的问题，事先、运行过程中和事后都可以不清楚对方的存在，因而是无关进程。应当说明的是，这种结构上无关的进程并不等于不存在联系，或关系。这些进程之间在逻辑上可能由于系统的原因存在着随机的联系或关系。

因而，在逻辑上，有结构上联系的进程之间可以有关系，不存在结构上联系的进程之间也可以存在关系。这后者就使得操作系统变得更复杂了，也正是下面所要说明的——进程之间的制约关系。有关这方面的详细内容及算法将在第 5 章讨论。

2. 逻辑上相关进程

正如前面所述，在逻辑上，有结构和无结构联系的进程之间都可以存在关系，这里重点强调的是进程之间的制约关系。所谓制约关系是指，系统中活动的进程之间，由于并发性，每个进程的推进可能受到系统内其他进程关于资源(软、硬件)、信息交换、通信等原因的“干扰”而暂停，或等待。

1) 直接制约关系

进程之间为了共同完成某项任务，或者协作完成，它们之间的联系是事先有所考虑，有意识的彼此相互“交换信息”。例如在图 3.3 中，如果分别将 I、C 和 P 都看成是进程，则 C 的“计算”过程只有等 I 完成，没有“数据”，C 是无法继续执行的。它们之间是一种协作关系，因而 I 和 C 的这种关系就可以看成是进程之间的直接制约关系。同样，C，P 之间也是这种关系。

2）间接制约关系

进程之间通过竞争系统某些资源产生的关系称为间接制约关系，它们之间是通过某种“媒介”。例如，两个用户进程 user1 和 user2 在运行过程中，都希望利用系统仅有的一台打印机（外部设备媒介）进行结果输出。由于打印机的物理特性决定了一个进程在使用时，其他进程是不能使用的（否则将导致打印结果杂乱无章，这是系统和用户都不能接受的），因而，假定 user2 当前正在使用打印机，而 user1 在某一时刻也希望进行打印输出，如果 user2 未打印完，user1 只有等待。问题所在就是，user1 和 user2 可能事先、运行过程中和事后都不清楚对方，本来没有任何关系，而由于竞争这种设备产生了间接关系。这种关系逻辑上可以发生在任意进程之间。

3.5 线程

在传统的操作系统中，进程的概念是操作系统结构的基础，进程模型是基于下面两个独立的概念：资源分配的单位和并发执行时调度的单位。

1. 资源分配的单位

由图 3.9 进程的组成以及系统对进程的控制和管理可知，一个进程拥有对资源的所有权，这些资源包括主存、I/O 通道、I/O 设备和文件等。操作系统实施保护功能，以防止进程之间发生可能的冲突。

2. 调度的单位

一个进程可能通过一个或多个程序（段）的执行轨迹执行，形成一条进程内执行流，或控制流，并且在执行过程中可能与其他进程并发交替进行。因此，一个处于活动空间的进程是操作系统调度的单位。

进程的这两个特点是相互独立的，操作系统将这两个属性分别赋予了两个不同的实体；拥有资源所有权的仍称为进程，而调度的单位称为线程，或轻量级进程。

3.5.1 线程的引入

“线”这个词本身可以表达两个含义：一个是轻，另一个是轨迹。因此在操作系统发展过程中，从计算机的效率、进程内的多个并发活动存在与控制的角度，也为了减少系统对于并发所带来的时/空开销，建立了线程（Thread）模型。

1. 线程

线程是进程内的一个相对独立的执行流，或控制流，是处理机分配的实体。虽然线程的概念是在分析了并发进程的活动之后才出现的，但是客观上它早已存在，随着并行技术的发展才逐渐明朗起来。传统的用户进程中只有一个执行流，所以传统进程都可以看成是单线程的。

由前面可知，一个进程是由 PCB 数据结构、程序和数据（结构）等组成，如果仅从 CPU 执行程序所必需的硬件现场来考察，只包含程序计数器、程序状态字、执行堆栈、通用寄存器组等。PCB 中的绝大多数条目与 CPU 执行程序代码是没有关系的。但由于进程也是资源拥有者，所以在实施进程创建、撤销、进程之间切换过程中，整个上下文都需要变化，尤其是要切换程序和数据的地址空间，这样系统开销较大。

有了线程的概念之后，进程模型得到了有效扩展，因为在一个进程中完全可以设置多个执行流程，对应多个线程。即在一个进程中可以同时运行多个不同的线程来执行不同的任务。

2. 多线程

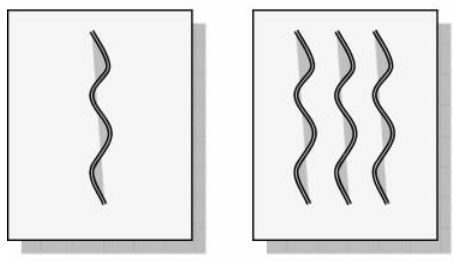
在许多应用当中，一些执行流之间具有内在的逻辑关系，涉及相同的代码和数据。如果将这些执行流放在同一个进程的框架之下，则这些执行流之间的切换便不涉及地址空间的变化。这样就可以在一个进程内根据应用的需要建立有多个线程，它们都共享该进程的状态和资源。也就是说它们驻留在同一个用户地址空间中，可以访问相同的数据，因而可以更好地实现多个任务间的合作。这样在一个进程内，线程的切换仅需改变寄存器和栈的现场，而包括程序和数据的地址空间可以保持不变。

同一个进程中的多个线程可以执行相同的代码段，此时对应同一个服务的多个请求。也可以执行不同的代码段，此时体现了逻辑上的合作关系，这些合作的线程可以利用共享的数据结构相互交换信息。

3.5.2　线程的结构与线程控制块 TCB

由于每个线程有自己的执行堆栈、程序计数器、通用寄存器组和状态标记，所以同样要为每个线程定义一个数据结构——线程控制块来包含这些数据成员，以实现系统对线程的管理和控制及线程之间的切换。由于同一个进程的多个线程共享同一地址空间，带来了系统时/空管理的改善。

图 3.19(a)和图 3.19(b)分别为进程与线程的示意图，以及线程控制块 TCB。



线程标识	线程号
处理机状态信息（现场）	通用寄存器
	指令计数器
	程序状态字
	栈指针
线程调度信息	线程状态
	线程优先数
	等待原因
	调度算法参数等

(a)　(b)

图 3.19　线程及线程控制块

引入线程的益处：

(1) 并发执行发生在线程实体上，因而系统对处理机的分配也发生在线程之间，而线程之间的切换改善了系统时空开销，包括内存管理、线程切换时间；

(2) 系统创建或终止一个线程的开销要比创建或终止一个进程的开销少得多；

(3) 线程之间通信的效率要高于进程之间的通信效率。进程间通信要内核介入，而同一个进程的多个线程由于共享同一地址空间，所以通信时无须内核介入。

图 3.20(a)和图 3.20(b)分别给出了进程和线程的模型。

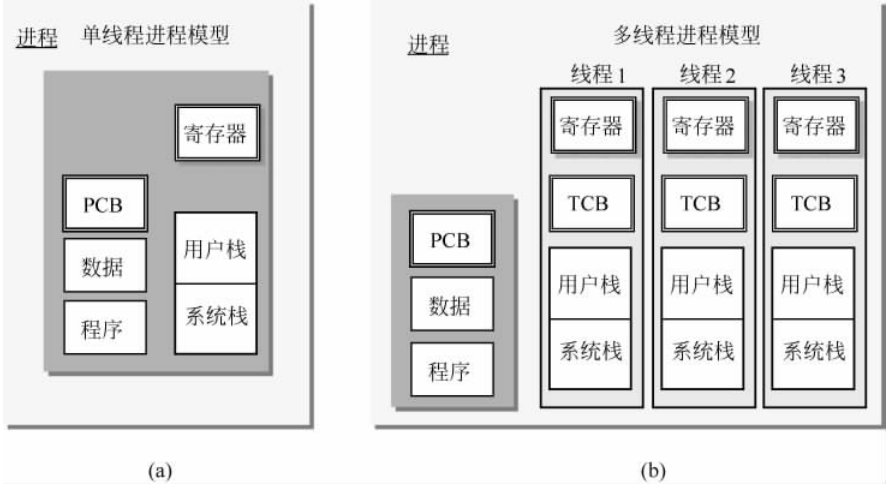


图 3.20 单线程和多线程的进程模型

3.5.3 线程与进程

一个线程可以创建和撤销另一个线程；同一个进程内的多个线程可并发运行。由于线程“继承”了进程并发的特性，因而线程同样具有进程的状态、状态转换。

(1) 调度。进程在传统的操作系统中，既是资源的拥有者，也是系统调度和分派的单位。而在引入线程之后，进程仅作为资源的分配单位，线程作为调度和分派的单位。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。

(2) 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，而且在一个进程内的多个线程之间，亦同样可以并发执行，因而使系统具有更好的并发性。更好的并发性也意味着系统资源利用率和系统吞吐量的提高。

(3) 系统资源。在引入线程的操作系统中，线程成为被调度和分派的基本单位。线程基本不拥有资源，只有一些运行所必需的资源(如程序计数器、一组寄存器和栈)，但它可以和进程内其他线程共享进程所拥有的全部资源，即一个进程的代码段、数据段以及系统资源(如打开的文件、I/O 设备等)。

(4) 系统开销。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，涉及当前进程的进程映像的保存，以及新选择进程的进程映像设置(恢复)。而线程的切换只需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，由于同一进程内多个线程具有相同的地址空间，因而线程之间的同步和通信也变得更加容易。

3.5.4 线程的实现

由于传统的原因，线程可分为内核级线程和用户级线程，也简称内核线程和用户线程。

1. 用户级线程

早期的线程都是用户级的，为了对用户级线程进行控制和管理，操作系统提供一个在用户空间执行的线程库。线程库是一组供所有应用程序所共享的应用级软件包。该线程库提供创建、调度、撤销线程功能，并在用户态完成。如图 3.22(a)所示，与线程有关的控制结构 TCB 保存在用户态空间，并由运行系统维护。用户线程运行在操作系统核心之上，但进程中的线程对内核是透明的，或者说内核无须知道它们的存在。操作系统内核只对进程进行管理。

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。用户及线程只使用用户堆栈和分配给所属进程的用户寄存器。当一个线程被派生时，线程库为其生成相应的线程控制块 TCB 等数据结构，置 TCB 中有关值并将该线程置于就绪状态。其处理过程与进程创建过程大致相似，不同的是：

(1) 用户级线程的调度算法只进行线程上下文切换，即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，新线程通过程序调用指针的变化使得程序计数器变化而得以执行；

(2) 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，即当一个进程由于 I/O 中断或时间片用完等原因造成该进程退出处理机，而属于该进程的执行中线程仍处于执行状态，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。

通过一个例子来阐述线程调度和进程调度的关系。假定进程 P 在它的线程 2 中运行，进程和作为进程部分的两个用户级线程的状态如图 3.21 所示，则可能发生以下任何一种情况。

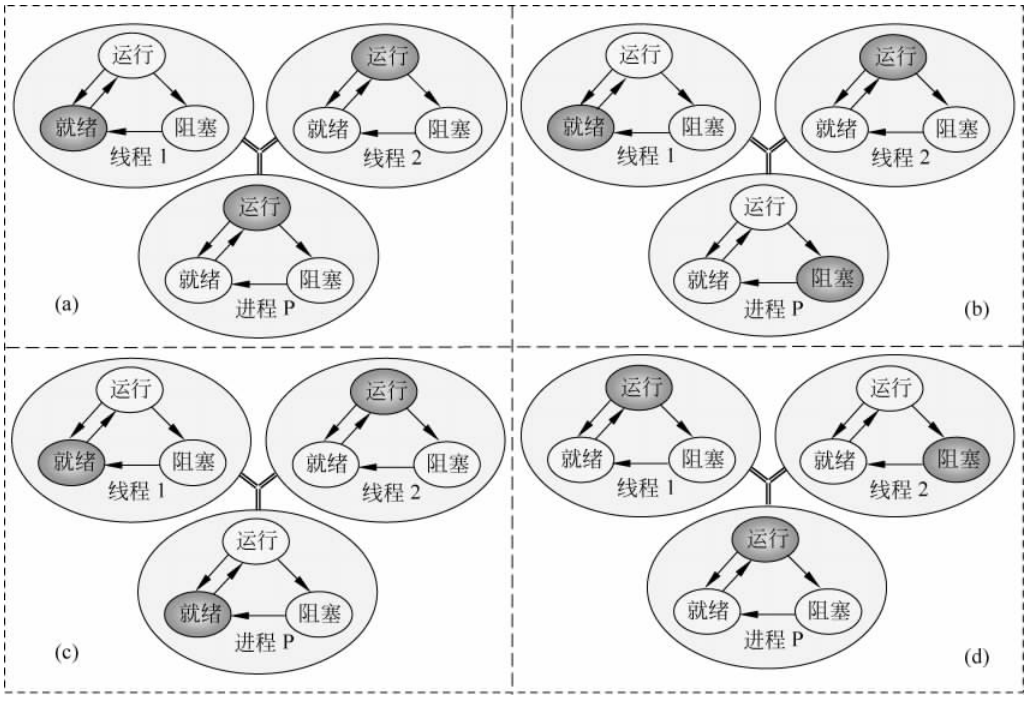


图 3.21　用户级线程状态与进程状态间的关系

(1) 线程 2 中执行的应用程序执行系统调用,阻塞了进程 P,例如进行一次 I/O。这导致控制权转移到内核,内核启动 I/O 操作,并切换到另一个进程。在此期间,根据线程库维护的数据结构,进程 P 线程 2 仍处于运行状态。值得注意的是,线程 2 的运行并不是真正意义上的被处理机执行,而是线程库察觉出它处于运行状态,见图 3.21(b)。

(2) 时钟中断把控制传递给内核,内核确定当前正在运行的进程 P 已经用完了它的时间片,内核将进程 P 置于就绪状态,并切换到另一个进程。在此期间,根据线程库维护的数据结构,进程 P 线程 2 仍处于运行状态。相应的状态图见图 3.21(c)。

(3) 线程 2 达到某处,它需要进程 P 的线程 1 所执行的某些动作。线程 2 进入阻塞状态,线程 1 从就绪状态转换到运行状态,进程自身保留在运行状态。相应的状态图见图 3.21(d)。

在第(1)种和第(2)种情况中,如图 3.21(b)和图 3.21(c)所示,当内核将控制切换回进程 P 时,恢复线程 2 中的执行。还需注意进程在执行线程库代码时被中断,或者是由于它的时间片用完了,或者是由于被一个高优先级的进程所抢占。因此在中断时,进程有可能处于线程切换的中间时刻,即正在从一个线程切换到另一个线程。当该进程被恢复时,将继续在线程库中执行,完成线程切换,并将控制权转移给该进程的一个新线程。

使用用户级线程代替内核线程有几个优点。

(1) 由于所有线程管理数据结构都在一个进程的用户地址空间中,因此,进程不需要为了线程管理而切换到内核模式,这避免了由用户模式→内核模式,再由内核模式→用户模式两种模式之间的切换,减少了系统开销。

(2) 线程之间的切换算法可以由应用程序确定。一个应用程序可能倾向于简单的轮转算法,另一个应用程序可能倾向于基于优先数算法,由于是应用程序内进行的线程切换,因而不会干扰内核调度。

(3) 用户级线程可以在任何操作系统中运行,不需要对低层内核进行修改以支持用户级线程的实现。

另外,用户级线程相对于内核级线程(见下一节)有两个明显的缺点。

(1) 在典型的操作系统中,许多系统调用都会引起用户执行流的阻塞。因此,当用户级线程执行一个系统调用时,不仅这个线程被阻塞,整个进程的所有线程都被阻塞了。

(2) 在纯粹的用户级线程策略中,由于内核是按进程作为调度单位的,因此一个多线程用户应用程序不能利用多处理机技术。内核一次只将一个进程分配给一个处理机,也就只能有一个线程可以执行。

2. 核心级线程

核心级线程由操作系统内核进行管理,线程的控制块 TCB 放在操作系统空间。线程的状态转换由操作系统完成。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,以使用户程序可以创建、执行、撤销线程。

与用户线程不同,核心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,核心级线程不会出现进程处于阻塞或等待状态,而线程处于执行状态的情况。

与用户级线程相比,核心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。核心级线程的模型见图 3.22(b)。

3. 混合线程

有些操作系统如 SUN Solaris 将用户线程与核心线程这两种方式结合起来，组成了混合线程，发挥各自的优势。在组合方式中，核心只知道核心线程，也只对它们实施调度。某些核心线程对应多个用户级线程，因此，产生了多对一，多对多的模型，如图 3.22(c)和图 3.22(d)所示。

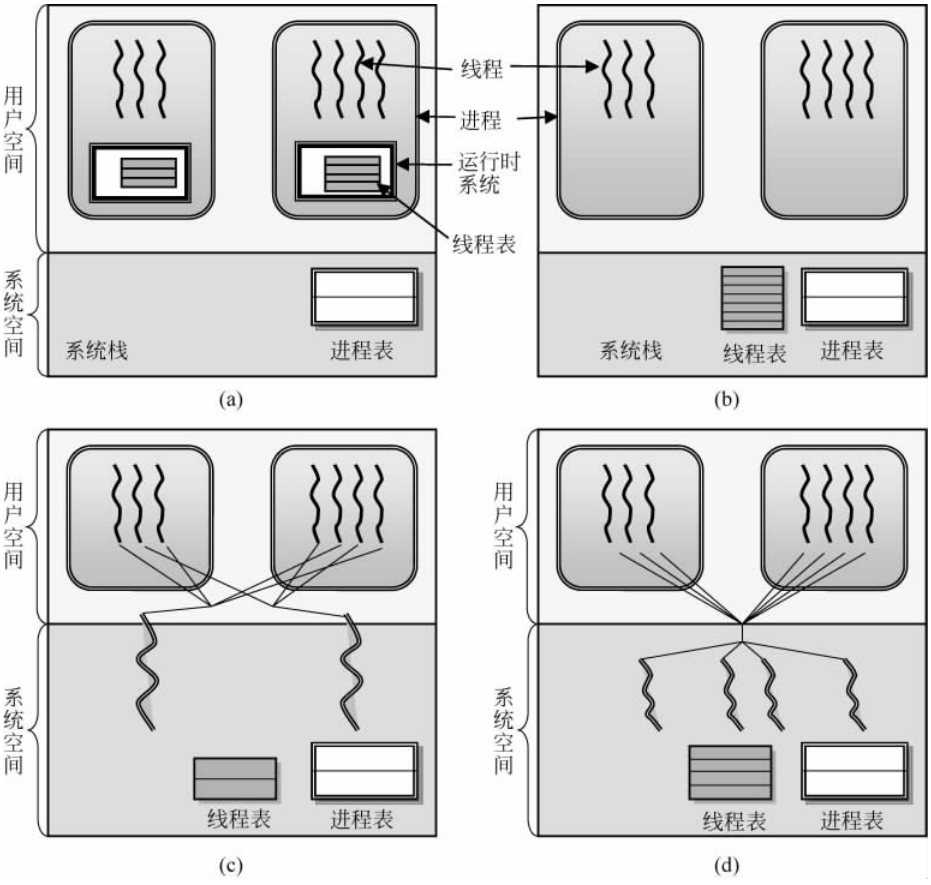


图 3.22　在用户空间和系统空间实现线程

3.5.5　线程的调度

现在由于线程既可以是用户线程，也可以是核心线程，就有了下面的问题。当进程可以有多个线程时，就存在两个层次的并发/并行：进程和线程。在进程可以有多个线程的系统中，调度是分情况的，或者说有本质的差别，即取决于所支持的是用户级线程，还是核心级线程(即操作系统内核感知什么)，还是既支持用户级线程，也支持核心级线程，下面简单地分两种情况进行说明。

1. 用户级线程

由于内核并不知道有线程存在，所以内核还是和仅有进程的情况一样地进行操作，选取一个进程，假设为 A，并给予 A 以时间片控制。A 中的线程调度再决定哪个线程运行，假设

为 A_1。由于多道线程并不存在时钟中断，所以这个线程可以按其意愿运行任意长的时间。如果该线程用完了系统给予进程 A 的全部时间片，内核就会选择另一个进程投入运行（假定就绪队列存在其他进程）。

在进程 A 终于又一次得到运行时，线程 A_1 会接着运行，该线程会继续耗费进程 A 的所有时间，直到它完成工作。不过，该线程的这种“不合群”的行为不会影响到其他进程。其他进程会得到应该得到的（系统）内核调度程序所分配的合适份额，不会牵扯到进程 A 内部发生的事。

现在考虑线程 A_i（$i=1,2,3$）每次 CPU 的工作比较少的情况，例如，在进程 A 50ms 的时间片中有 5ms 的计算工作。于是，每个线程 A_i 运行 5ms，然后将 CPU 交还给线程调度程序。这样在内核切换到进程 B 之前，就会有 A_i 的运行序列：$A_1, A_2, A_3, A_1, A_2, A_3, A_1, A_2, A_3, A_1$（假定线程调度的时间耗费忽略不计），这种情形如图 3.23(a)所示。

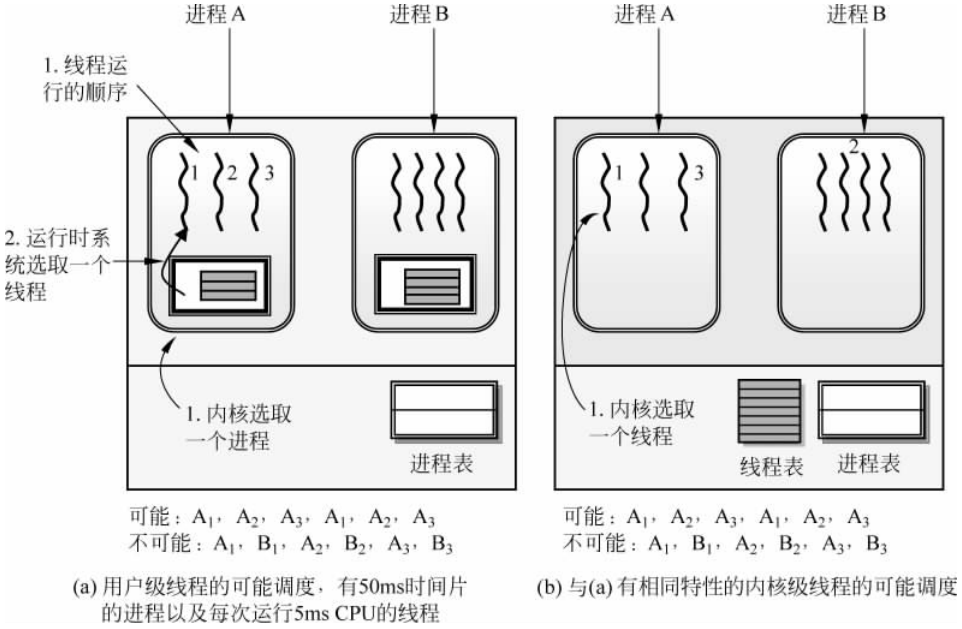


图 3.23 用户级线程和核心级线程

2. 核心级线程

现在考虑使用内核级线程的情形。内核选择一个特定的线程运行，它不用考虑该线程属于哪个进程（不过如果有必要的话，也可以考虑）。对被选择的线程赋予一个时间片，而且如果超过了给定的时间片，就会强制挂起该线程。一个线程在 50ms 的时间片内，5ms 之后被阻塞。在 30ms 的时间段中，线程的顺序可以是 $A_1, B_1, A_2, B_2, A_3, B_3$，如图 3.23(b)所示。在这种参数和用户级状态下，有些情形是不可能出现的。

用户级线程和内核级线程之间的差别在于性能。用户级线程的线程切换需要少量的机器指令，而内核级线程需要完整的上下文切换，修改内存映像，使高速缓存失效，这就导致了若干数量级的延迟或时间代价。另一方面，在有内核级线程时情况下，一旦线程被阻塞在 I/O 上，就不需要像用户级线程中那样将整个进程阻塞（挂起）。

从进程 A 的一个线程切换到 B 的一个线程，其代价高于运行进程 A 的第二个线程（因为必须修改内存映像，清除内存高速缓存的内容），内核对此是了解和掌握的，并可运用这些信息做出决定。例如，若有两个在其他方面同等重要的线程在就绪队列（只是所属进程不同），此时，其中属于同一进程的一个线程刚好被阻塞，而另一个线程属于其他进程，那么系统倾向于前者是有理由的（代价小）。

另一个重要因素是用户级线程可以使用专为应用定制的线程调度程序。例如，假设一个工作线程刚刚被阻塞，而用于调度的分派线程和其他的工作线程都是就绪的，那么应该运行哪一个呢？由于运行时系统了解所有线程的作用，所以会直接选择分派线程接着运行，这样分派线程就会从其他的工作线程中启动一个运行。在一个工作线程经常组塞在 I/O 上的环境中，这种策略能将并行/并发/度最大化。

在内核级线程中，内核是不了解每个线程的作用（虽然可以赋予它们不同的优先级）。不过，一般而言，应用定制的线程调度程序能够比内核更好地满足用户的需求。

3.5.6 线程的应用

在利用计算机解决问题的各种应用当中，许多应用都是同时发生着多种活动，并且在逻辑上涉及多个控制流，如数据库应用、信息检索以及各种具有前、后台处理特性的任务。

这些应用当中的控制流存在着内在的并发性，一些控制流被阻塞了，另外一些控制流可以继续。

从系统的角度，多进程可实现多控制流，但由于进程的特性决定了这些控制流的活动与一个应用任务内的多个控制流的活动是不同的，很难设想两个不同的用户可同时在编辑同一个文档。由于线程的空间就处在进程之内，因而面对一个应用的进程，可利用线程实现多个活动过程，即多个控制流。下面两个例子采用线程是十分有益的。

1. 字处理程序多线程应用

考虑一个字处理程序。通常它所编辑的文档在屏幕上的形式与打印的格式是精确对应的。

假如一个用户正在编写一本书。从编辑的角度，最容易的方法就是将整本书作为一个文件，这样，查询、全局替换都非常容易。当然，也可以将每一章都处理成单独的一个文件，特别是将每个小节和子小节都分成单个文件，但在对全书进行全局修改时，那种麻烦是可以想象的。因为有成百个文件必须被一个一个地编辑。

现在考虑，如果在一个有 800 页文件的第 1 页上删掉了一个句子，在检查了所修改的页面，并确认是正确的之后，马上打算在第 600 页上进行另一个修改（可能要查阅只在那里出现的一个短语），并输入一条命令通知字处理程序转到该页面。这时字处理程序被强制对整本书的前 600 页重新进行格式处理，因为原第 1 页已删去一些字符，因而字处理程序并不知道第 600 页的第 1 行现在在什么位置。而在第 600 页可以真正在屏幕上显示出来之前，计算机可能要拖延相当一段时间，如图 3.24(a)所示，从而令用户不甚满意。

多线程在这里可以发挥作用。假设字处理程序被编写成含有两个线程的程序。一个线程与用户交互，而另一个在后台重新进行格式处理。一旦在第 1 页中的语句被删掉，交互线程就立即通知格式化线程对整本书重新进行处理。同时，交互线程继续监控键盘和鼠标，并响应诸如滚动第 1 页之类的简单命令，此刻，另一个线程正在后台疯狂地“运算”。这样就有

可能使得重新格式化赶在用户请求查看第 600 页之前完成，第 600 页立刻可以在屏幕上显示处理。

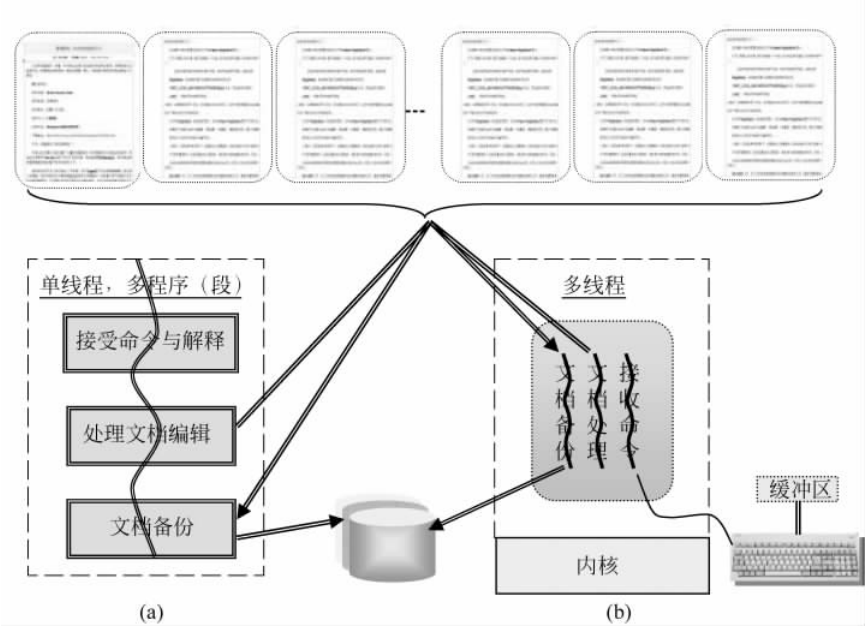


图 3.24 有 3 个线程的字符处理程序

考虑到许多字处理程序都有每隔若干分钟自动在磁盘上保存整个文件的特点，用于避免由于程序崩溃、系统崩溃或电源故障而造成用户一整天工作丢失的情况，再增设第 3 个线程用于处理磁盘备份，而不会干扰其他两个线程。拥有 3 个线程的情形见图 3.24(b)。

如果程序是单线程的，那么在进行磁盘备份时，来自键盘和鼠标的命令就会被忽略，直到备份工作完成，这样用户会认为性能很差。

很显然，在这里如果是 3 个不同的进程，则不可能共同完成这个工作，因为所有的 3 个进程都需要在各自的文件上工作(有自己独立的活动空间)。通过让 3 个线程代替 3 个进程，三个线程共享内存，于是它们都可以访问同一个正在编辑的文件。

许多交互式的应用程序也存在类似的情形。例如调用数据库数据表应用，其中数据汇总、排序以及查询就可以通过设置多个线程实现其并发运行，即在数据汇总时，可接收用户的查询命令等，这样会使得用户感觉这些事件在同时发生，没有停顿的感觉。

多线程应用的另一个例子是大家很熟悉的，浏览器的应用。使用浏览器，可以在下载图片的同时滚动页面；在访问新页面时，播放动画和声音、打印文件等。

2. 万维网上服务器的多线程应用

现在考虑另一个多线程发挥的例子，在万维网上用户对页面的请求发送给服务器，而所请求的页面发送给客户机/端。在多数的 Web 站点上，某些页面较其他页面有更多的访问。例如，用户在网上了解摄录像机时，对 Sony 站点主页的访问就可能远远超过对深藏在页面树里的某个摄录像机技术说明书页面的访问。基于这样的情况，Web 服务器可以将获得大量访问的页面的集合保存在主存中，避免到磁盘上去调这些页面，从而改善访问服务质量和

性能。这样的一种页面集合通常称为高速缓存(当然,高速缓存也运用在许多其他的场合)。

一种组织 Web 服务器的方式如图 3.25 所示。这里一个称为分派程序从网络中读入工作请求。在检查请求之后,分派线程挑选一个空转的(即被阻塞的)工作线程(提交该请求通常是在每个线程所配有的某个专门字中写入一个消息指针),接着分派线程唤醒休眠(意味阻塞)的工作线程,将其从阻塞状态转变为就绪状态。

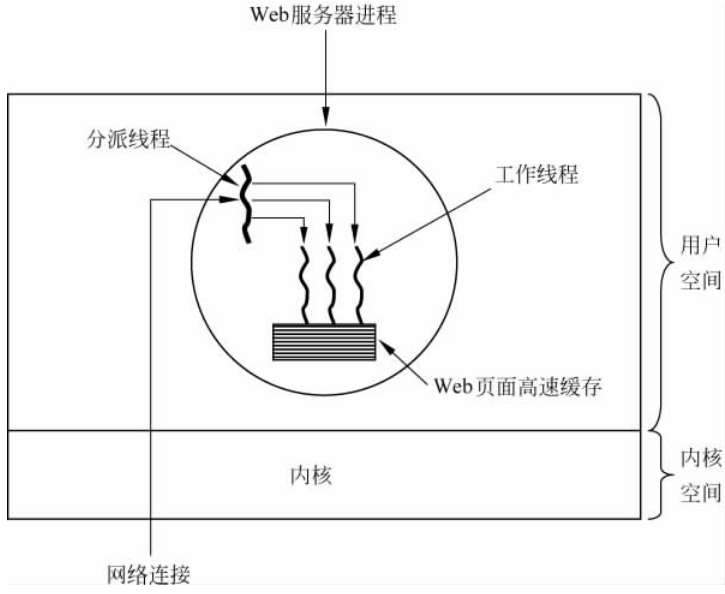


图 3.25 一个多线程的 Web 服务器

在工作线程被唤醒之后,它检查有关的请求是否在 Web 页面高速缓存之中。这个高速缓存是所有线程都可以访问的。如果没有,该线程开始从一个磁盘调入页面的读操作,并且被阻塞直到该磁盘操作完成。

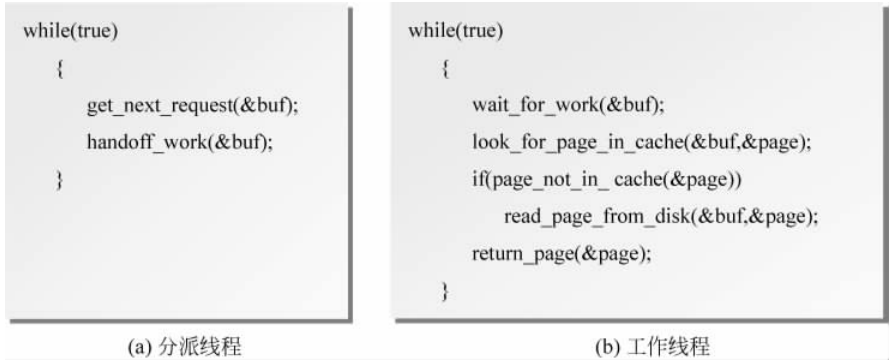
当上述线程被阻塞在磁盘操作上时,为了完成更多的工作,分派线程可能挑选另一个线程投入运行,也可能是另一个当前就绪的工作线程投入运行。

这种模型允许把服务器编写为顺序线程的一个集合。在分派线程的程序中,包含一个无限循环,该循环用来获得工作请求,并且将工作请求派给工作线程。每个工作线程的代码包含一个从分派线程接收请求,并且检查 Web 页面高速缓存中是否存在所需页面的无限循环。如果在 Web 页面高速缓存中,就将该页面返回用户(客户机),接着该工作线程阻塞,等待一个新的请求。如果没有在高速缓存中,工作线程从一个磁盘调入该页面,将该页面返回用户,然后工作线程阻塞再等待一个新的请求。

图 3.26 给出了代码的大致框架,其中,buf、page 分别是保存工作请求和 Web 页面的相应结构。

3. 网络计算与多线程应用

网络计算基于并行计算和分布式系统,其主要思想是将应用根据所需处理类型划分成多个半独立的部分,继而将其分配给网络中不同的结点(独立的计算机),此计算方式可以最大限度地利用不同体系结构的独特能力和大型组织中的空闲资源。因而一些未使用的



```
while(true)
    {
        get_next_request(&buf);
        handoff_work(&buf);
    }
```

(a) 分派线程

```
while(true)
    {
        wait_for_work(&buf);
        look_for_page_in_cache(&buf,&page);
        if(page_not_in_ cache(&page))
            read_page_from_disk(&buf,&page);
        return_page(&page);
    }
```

(b) 工作线程

图 3.26 对应图 3.25 的代码框架

CPU 周期可在一段活跃期之后的不活跃期中被利用。网络计算语境下的客户/服务器系统、PC 和工作站机群、网格计算和云计算等往往涉及多处理机系统，也因此与多线程机制密切相关。

类似图 3.25 的 Web 应用，另一经典的应用是基于客户/服务器的分布式数据库系统，多个客户发送查询请求给一个能访问数据库的服务器。该服务器代表客户执行查询，然后发送各自结果回应每个客户。这里，多线程的进程就是一种提供服务器应用的有效方式。通常，服务器进程服务于多个客户，而每个客户的请求触发服务器中一个新线程的创建。还有一些典型的应用，如服务器中的文件管理或通信控制、前后台处理、异步处理等。

应当说明的是，并不是在所有的计算机系统中线程机制都是适用的，对于很少做进程调度和切换的系统（如实时系统），由于任务的单一性，设置线程相反会占用更多的内存空间和寄存器。由此，可以得出结论：最适合使用线程的系统是多处理机系统。

最后需要指出的是，现代编程语言如 Java 内嵌了对多线程的支持，类 Thread 是标准库中的一个类，它支持一些处理线程的方法。例如 start 方法派生出一个新的控制线程，它能通过调用方法 stop 或 suspend 停止或挂起线程。类 Thread 也支持由 start 调用的方法 run 来激活线程。实际上，Thread.run 的标准实现并不做任何事，但可以扩展类 Thread 提供新的 run 方法。若底层（云）操作系统支持多线程机制，Java 线程通常就能映射到实际操作系统的线程。因此使用 Java 编写的应用能在多处理器环境中并行执行。显而易见，这对当下流行的网络计算环境（客户/服务器系统、PC 和工作站机群、网格计算和云计算等）而言也至关重要。

3.6 Linux 进程管理

Linux 是多用户、多任务的操作系统，这就意味着多个用户可以同时使用一个操作系统，而每个用户又可以同时运行多个命令。在这样的系统中，各种计算机资源（如文件、内存、CPU 等）的分配和管理都以进程为单位。为了协调多个进程对这些共享资源的访问，操作系统要跟踪所有进程的活动，以及它们对系统资源的使用情况，实施对进程和资源的动态管理。

3.6.1 Linux 进程结构

1. Linux 进程的组成

在 Linux 系统中，进程由 3 部分组成。

(1) 正文段(Text)：存放被执行的机器指令。这个段是只读的，它允许系统中正在运行的两个或多个进程之间能够共享这一代码。

(2) 用户数据段(User Segment)：存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里包含的信息可以被改变。虽然进程之间可以共享正文段，但是每个进程需要有它自己的专用用户数据段。

(3) 系统数据段(System Segment)：该段有效地存放程序运行的环境。事实上，这正是程序和进程的区别所在，是进程实体最重要的一部分。之所以说它有效地存放程序运行的环境，是因为这一部分存放有进程的控制信息。系统中有许多进程，操作系统要管理它们、调度它们运行，就是通过这些控制信息。Linux 为每个进程建立了 task_struct 数据结构来容纳这些控制信息——PCB，也被称作进程描述符(Process Descriptor)。

任务(Task)和进程(Process)是两个相同的术语，task_struct 其实就是 PCB。task_struct 容纳了一个进程的所有信息，不仅是系统对进程进行控制的唯一手段，也是系统实现进程调度的主要依据。

当一个进程被创建时，Linux 为每个新创建的进程动态地分配一个 task_struct 结构。当进程运行结束时，系统撤销该进程的任务结构体。进程的任务结构体是进程存在的唯一标志。Linux 在内存空间中开辟了一个专门的区域存放所有进程的任务结构体，即在操作系统的内核空间设置了一个 task 数组，该数组的每一个元素是一个指向任务结构体的指针，所以 task 数组又称为 task 向量。

```
struct  task_struct  * task[NR_TASKS] = {&init_task};
```

这个数组的大小默认为 512，表明在 Linux 系统中能够同时运行的进程最多可有 512(实际上与内存的大小有关)。

Linux 支持两种进程：普通进程和实时进程。实时进程具有一定程度上的紧迫性，要求对外部事件做出非常快的响应；而普通进程则没有这种限制。这两种进程的区分也反映在 task_struct 数据结构中了。

Linux 中包含进程所有信息的 task_struct 数据结构是比较庞大的，大约 1K 多字节，包含了许多字段/类别，按照字段，可分成如下几类。

(1) 标识号。系统通过进程标识号唯一识别一个进程，但进程标识号并不是进程对应的 task_struct 结构指针在 task 数组中的索引号。另外，一个进程还有自己的用户和组标识号，系统通过这两个标识号判断进程对文件或设备的访问权。

(2) 状态信息。一个 Linux 进程可有如下几种状态：运行、等待、停止和僵死。

(3) 调度信息。调度程序利用该信息完成进程之间的切换。

(4) 有关进程间通信的信息。系统利用这一信息实现进程间的通信。

(5) 进程链信息。在 Linux 系统中，除初始化进程之外，任何一个进程都具有父进程。每个进程都是从父进程中“克隆”出来的。进程链/家族链则包含进程的父进程指针、和该进

程具有相同父进程的兄弟进程指针以及进程的子进程指针。另外，Linux 利用一个双向链表记录系统中所有的进程，这个双向链表的根就是 init 进程。利用这个链表中的信息，内核可以很容易地找到某个进程。

(6) 时间和定时器。系统在这些字段中保存进程的建立时间，以及在其生命周期中所花费的 CPU 时间，这两个时间均以 jiffies 为单位。该时间由两部分组成，一是进程在用户模式下花费的时间，二是进程在系统模式下花的时间。Linux 也支持和进程相关的定时器，应用程序可通过系统调用建立定时器，当定时器到期，操作系统会向该进程发送 sigalrm 信号。

(7) 文件系统信息。进程可以打开文件系统中的文件，系统需要对这些文件进行跟踪。系统使用这类字段记录进程所打开的文件描述符信息。另外，还包含指向虚拟文件系统(Virtual File Systems，VFS)两个索引结点的指针，这两个索引结点分别是进程的主目录以及进程的当前目录。索引结点中有一个引用计数器，当有新的进程指向某个索引结点时，该索引结点的引用计数器会增加计数。未被引用的索引结点的引用计数为 0，因此，当包含在某个目录中的文件正在运行时，就无法删除这一目录，因为这一目录的引用计数大于 0。

(8) 和进程相关的上下文信息。如前所述，进程可被看成是系统状态的集合，随着进程的运行，这一集合发生变化。进程上下文就是用来保存系统状态的 task_struct 字段(相当于 CPU 现场)。当调度程序将某个进程从运行状态切换到暂停状态时，会在上下文中保存当前的进程运行环境，包括 CPU 寄存器的值以及堆栈信息。当调度程序再次选择该进程运行时，则会从进程上下文信息中恢复进程的运行环境。

图 3.27 给出了其简要结构描述。由该 task_struct，可以看出，系统管理每个进程的信息中包括指向相应管理的指针，如进程队列、内存管理、文件结构指针等。

2. 进程空间和系统空间

系统为每个进程分配一个独立的虚拟地址空间(虚拟内存)。进程的虚拟地址空间被分做两个部分：用户空间和系统空间。

用户进程本身的程序和数据(可执行映像)映射到用户空间中。进程空间中还有进程运行用户程序时使用的堆栈，称为进程堆栈。系统对这个进程进行控制和管理的信息，如进程控制块等，也映射到进程空间。

内核堆栈也在进程空间中。内核被映射到所有进程的系统空间中。它们只允许在具有较高特权的核心态下访问。进程运行在特权较低的用户态下时，不允许它直接访问系统空间，如图 3.28 所示。进程只能通过系统调用换为核心态后，才能访问系统空间。一个进程在运行过程中，总是在两种执行状态之间不断地转换。

实际上，进程拥有两个栈，用户模式栈与核心模式栈，分别在相应模式下使用。进程描述符(进程控制块)和进程核心栈的空间分配在一起，内核为它们分配两个连续的物理页面。

```
union task_union {
        struct task_struct task;
        unsigned long stack[2048];
};
```

因为进程控制块/描述符已经占用了 1KB 多的空间，所以核心栈的有效空间是 6KB 多一点，合理的设计使得这个容量已经足够了。核心栈与进程描述符如图 3.29 所示。

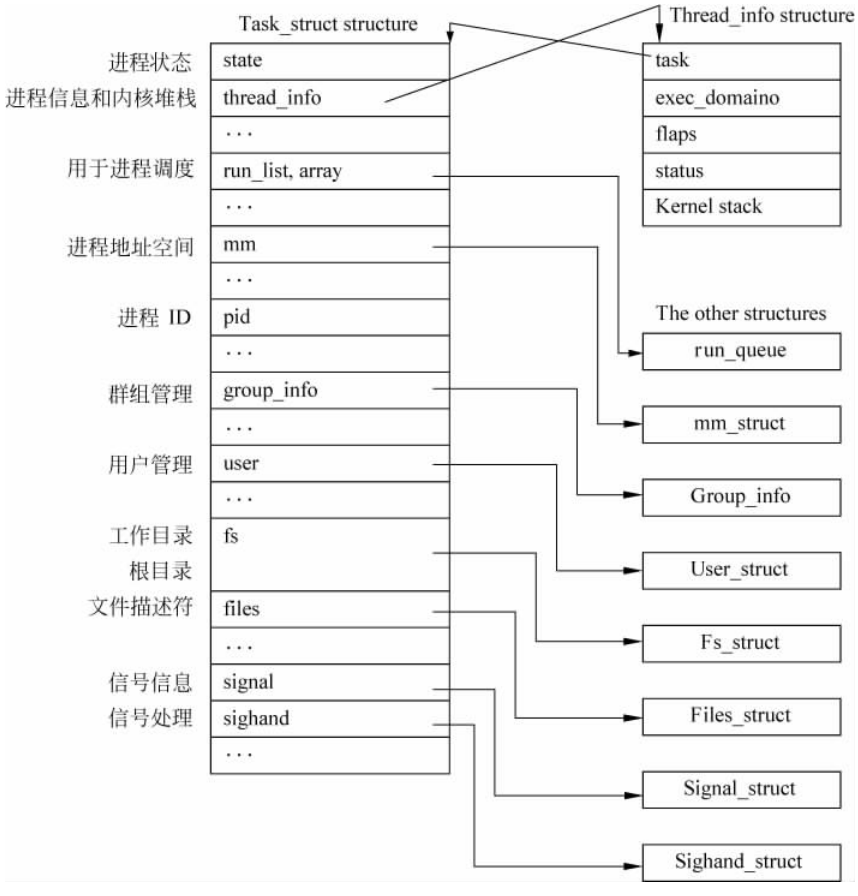


图 3.27　task_struct 结构

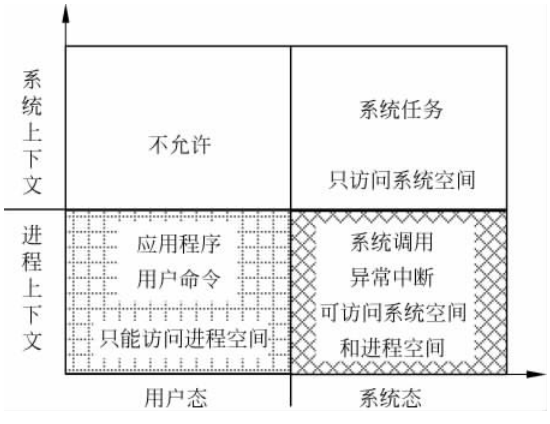


图 3.28　执行状态和上下文

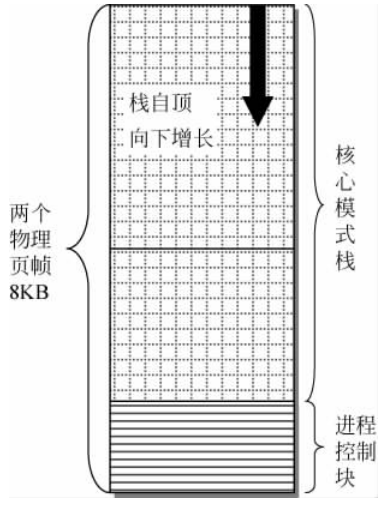


图 3.29　核心栈与进程描述符

3. 进程上下文和系统上下文

把系统提供给进程的处于动态变化的运行环境总和称为进程上下文。系统中的每一个进程都有它自己的上下文。进程因时间片用完或因等待某个事件而阻塞时，进程调度需要把 CPU 的使用权从当前进程交给另一个进程，这个过程称为进程切换（Process Switching）。进程的切换又称为上下文切换（Context Switching）。在系统内核为用户进程服务，例如进程执行一个系统调用时，进程的执行状态要从用户态转换为核心态。但是，此时内核的运行仍是进程的一部分，所以说这时内核是运行在进程上下文中。系统在完成自身任务时的运行环境称为系统上下文（System Context）。内核在系统上下文中执行时不会阻塞。

3.6.2 Linux 进程状态及运行模式

1. 进程的状态

进程的状态是调度和切换的依据，Linux 的进程状态如表 3.7 所示，状态转换图如图 3.30 所示。

表 3.7 Linux 进程的状态

状态的内核表示	含 义
TASK_RUNNING	可运行
TASK_INTERRUPTIBLE	可中断的等待状态
TASK_UNINTERRUPTIBLE	不可中断的等待状态
TASK_ZOMBIE	僵死
TASK_STOPPED	暂停

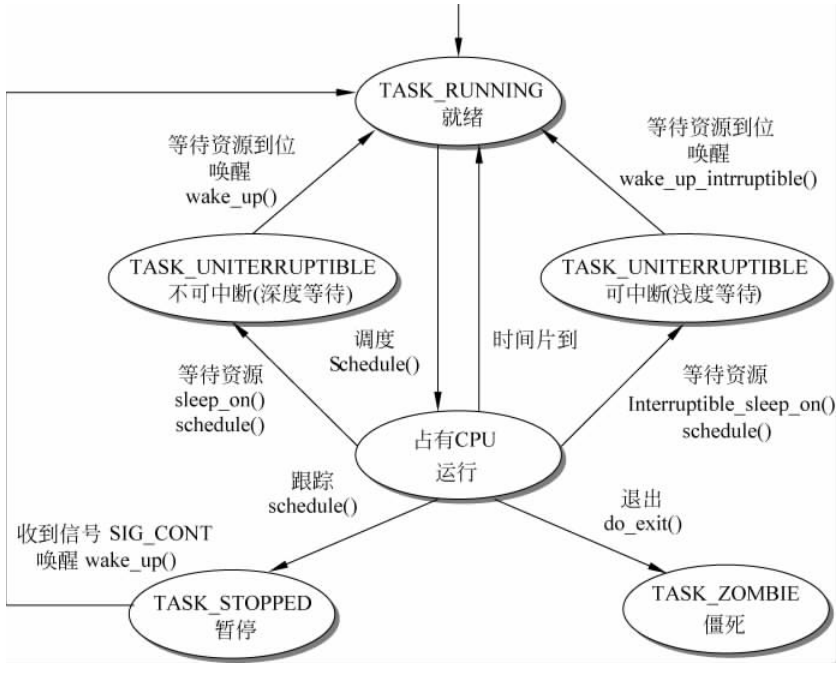


图 3.30 Linux 进程状态转换

Linux 进程的状态由进程描述符(PCB)task_struct 中的数据项 state 表示。进程状态 state 的取值有如下几种。

```
#define TASK_RUNNING                0
#define TASK_INTERRUPTIBLE          1
#define TASK_UNINTERRUPTIBLE        2
#define TASK_ZOMBIE                 4
#define TASK_STOPPED                8
```

(1) 可运行状态——处于这种状态的进程,要么正在运行,要么正准备运行。正在运行的进程就是当前进程(由 current 所指向的进程),而准备运行的进程只要得到 CPU 就可以立即投入运行,CPU 是这些进程唯一等待的系统资源。系统中有一个运行队列(run_queue),用来容纳所有处于可运行状态的进程,调度程序执行时,从中选择一个进程投入运行。

(2) 等待状态——处于该状态的进程正在等待某个事件或某个资源,它肯定位于系统中的某个等待队列(wait_queue)中。Linux 中处于等待状态的进程分为两种:可中断的等待状态和不可中断的等待状态。处于可中断等待态的进程可以被信号唤醒,如果收到信号,该进程就从等待状态进入可运行状态,并且加入到运行队列中,等待被调度;而处于不可中断等待态的进程是因为硬件环境不能满足而等待,例如等待特定的系统资源,它在任何情况下都不能被打断,只能用特定的方式来唤醒它,例如唤醒函数 wake_up()等。

(3) 暂停状态。此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 信号后就处于这种状态。例如,正接受调试的进程就处于这种状态。

(4) 僵死状态。进程虽然已经终止,但由于某种原因,父进程还没有执行 wait()系统调用(一种同步),终止进程的信息也还没有回收。顾名思义,处于该状态的进程就是死进程,必须进行相应处理以释放其占用的资源。

有关 Linux 系统的调度将在第 4 章讨论。

2. 进程的模式和类型

在 Linux 系统中,进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或者内核之外的系统程序,那么对应进程就在用户模式下运行;如果在用户程序执行过程中出现系统调用或者发生中断事件,就要运行操作系统(即核心)程序,进程模式就变成内核模式。在内核模式下运行的进程可以执行机器的特权指令;而且,此时该进程的运行不受用户的干预,即使是 root 用户也不能干预内核模式下进程的运行。

按照进程的功能和运行的程序分类,进程可划分为两大类:一类是系统进程,只运行在内核模式,执行操作系统代码,完成一些管理性的工作,例如内存分配和进程切换;另外一类是用户进程,通常在用户模式中执行,并通过系统调用或在出现中断、异常时进入内核模式。用户进程既可以在用户模式下运行,也可以在内核模式下运行,如图 3.31 所示。

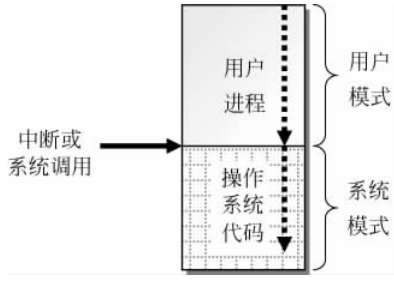


图 3.31　用户进程的两种运行模式

3. Linux 线程

线程是和进程紧密相关的概念。一般说来，Linux 系统中的进程应具有一段可执行的程序、专用的系统堆栈空间、私有的 PCB(即 task_struct 数据结构)和独立的存储空间。Linux 系统中的线程只具备前 3 个组成部分，而缺少自己的存储空间。

Linux 的内核线程和其他操作系统的内核实现不同。大多数操作系统单独定义线程，从而增加了内核和调度程序的复杂性。而 Linux 则将线程定义为“执行上下文”，实际只是进程的另外一个执行上下文而已，没有真正意义上的线程概念(见图 3.27)。这样，Linux 内核只需区分进程，只需要一个进程/线程数组，而调度程序仍然是进程的调度程序。

3.6.3 Linux 进程控制

1. 进程创建

系统启动时总是处于核心模式，此时只有一个进程：初始化进程。像所有进程一样，初始化进程也有一个由堆栈、寄存器等表示的机器状态。当系统中有其他进程被创建并运行时，这些信息将被存储在初始化进程的 task_struct 结构中。在系统初始化的最后，初始化进程启动一个核心线程(init)然后保留在 idle 状态。如果没有任何事要做，调度管理器将运行 idle 进程。idle 进程是唯一个不是动态分配 task_struct 的进程，它的 task_struct 在核心构造时静态定义并且名字叫 init_task。

由于是系统的第一个真正的进程，所以 init 核心线程(或进程)的标志符为 1。它负责完成系统的一些初始化设置任务(如打开系统控制台与安装根文件系统)，以及执行系统初始化程序，如/etc/init、/bin/init 或者/sbin/init ，这些初始化程序依赖于具体的系统。init 程序创建系统中的新进程，这些新进程又创建各自的新进程。例如 getty 进程将在用户试图登录时创建一个 login 进程。系统中所有进程都是从 init 核心线程中派生出来。

Linux 系统创建进程的方法很特别，新进程通过“克隆”/复制老进程或当前进程来创建。通常利用系统调用 fork 或 clone 创建一个新进程，然后新进程通过调用 exec 系列函数执行真正的任务。函数 fork()调用成功，当前进程就拥有了一个子进程。该函数返回两个值，其中子进程返回 0，父进程返回的是子进程的 pid 值。复制发生在核心状态下的核心中。系统从物理内存中分配出来一个新的 task_struct 数据结构，同时还有一个或多个包含被复制进程堆栈(用户与核心)的物理页面。然后创建唯一地标记此新任务的进程标志符。新创建的 task_struct 将被放入 task 数组中，另外将被复制进程的 task_struct 中的内容页表复制到新的 task_struct 中。进程创建的大致过程如下：

(1) 为新进程分配任务结构体内存空间；

(2) 把父进程任务结构体复制到子进程的任务结构体；

(3) 为新进程在其虚拟内存建立内核堆栈；

(4) 对子进程任务结构体中部分进行初始化设置；

(5) 把父进程的有关信息复制给子进程，建立共享关系；

(6) 把子进程的 counter 设为父进程 counter 值的一半；

(7) 把子进程加入到可运行队列中；

(8) 结束 fork()函数返回 PID 值。

一个创建子进程的例子与图 3.32 描述了在 Linux 系统中创建进程的过程和特点。子

进程在被创建后执行的是父进程的程序代码。父进程执行 fork()返值是子进程的 PID 值，子进程执行 fork()的返值是 0。创建子进程的例子如下：

```
#include <sys/typex.h>
#include <unisd.h>
main()
{
    pid_t val;
    printf("PID before fork(): %d\n",(int)getpid());
    if(val = fork())
        printf("parent process PID: %d\n",(int)getpid());
    else
        printf("child process PID: %d\n",(int)getpid());
}
```

该程序的执行结果：

```
PID before fork(): n
parent process PID: n
child process PID: n + 1
```

其中，n 为当前系统执行两个进程时，系统当时的 PID 值(一个十进制数)。

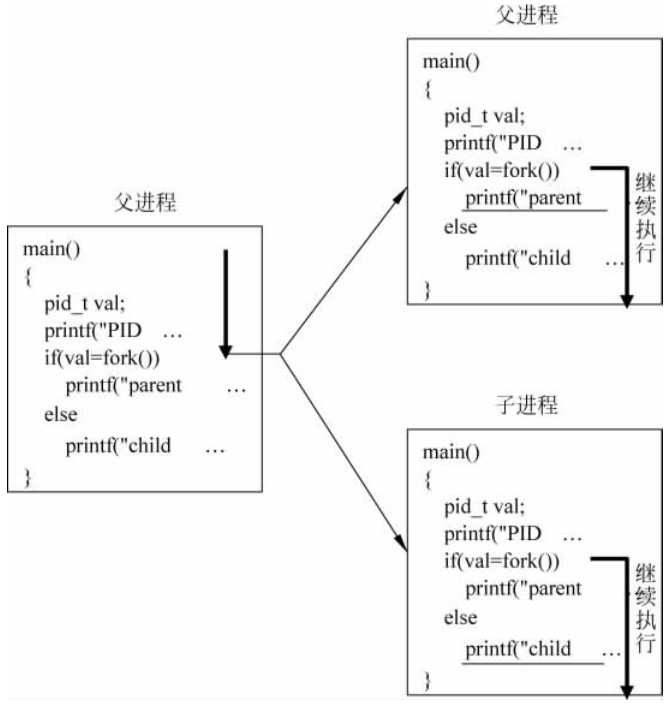


图 3.32 父进程创建子进程及并发执行示例

2. 进程的等待与终止

父进程发出 fork()系统调用创建一个子进程是为了通过子进程来完成某项任务，因此，父进程在创建一个子进程后，通常等待子进程运行终止。父进程等待子进程终止是通过系统调用 wait()来实现。而进程的终止可以通过系统调用 exit()终止自己。

习题 3

3.1 多道程序可以提高系统效率，是否在内存中尽可能多地存放多个程序，为什么？

3.2 操作系统通过什么概念来刻画程序的并发执行、资源分配及随机性？

3.3 试画出下面 5 条语句的前驱图：

S_1：x=5；S_2：y=x+8；S_3：z=x+y；S_4：a=x+y+z；S_5：b=y+a。

3.4 从多个程序在单处理机上执行的角度来考察，并发的含义是什么？

3.5 有下面的 5 条语句，试画出前驱图；

S_1：a=x+10；S_2：b=a+10；S_3：c=4 * x；S_4：d=b+c；S_5：e=d+5；并根据 Bernstein 条件，证明 S_2 和 S_3 是可以并发执行的，而 S_4 和 S_5 语句是不能并发执行的。

3.6 在书中给出的有关进程描述中，它们所具有的本质是什么？

3.7 并发运行因为什么发生结果不可再现性？

3.8 进程的基本特征是什么？

3.9 进程的构成中，PCB 是属于系统还是用户的部分，它的作用是什么？

3.10 为什么说 PCB 是操作系统感知进程存在的唯一标志？

3.11 用户进程所执行的程序一定是用户自己编写的应用程序，这句话是否正确？

3.12 进程的 3 个基本状态是什么？怎样从进程状态的角度理解进程的“走走停停”？

3.13 进程如果正常执行结束，为什么还要在系统中设置一个完成状态，而不直接从系统中退出消亡？

3.14 有进程状态转换（变迁）如图 3.33 表示，图 3.33 中的数字表示一种形式的状态转换。试判别下述诸条件是否成立？为什么？

1 → 2；1 → 3；2 → 1；

2 → 4；3 → 1；3 → 4；

4 → 1；4 → 2；4 → 3；

5 → 1；5 → 4；

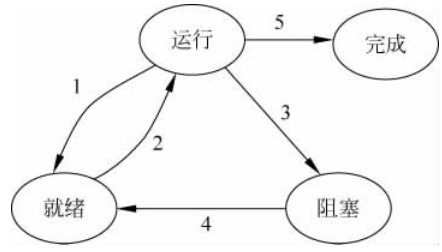


图 3.33 题 3.14

其中，条件式 x → y 表示若一进程产生状态转换 x，则导致另一个进程产生状态转换 y(x,y=1,2,3,4,5)。

3.15 引入挂起状态的理由是什么，给出一个挂起的时机（任意就绪、阻塞活动状态下）。

3.16 为什么可以出现从各种状态的进程直接进入完成状态？

3.17 操作系统为什么需要考虑处理机的执行模式，通常由用户模式进入系统模式的时机是什么？

3.18 解释原语的含义，如果原语是由一段程序构成，为什么说原语是不能并发执行的？不是说系统内多个程序（进程）可以并发执行吗？

3.19 给出一个唤醒进程的时机，并说明可以由谁唤醒一个在阻塞态的进程。

3.20 进程之间的制约关系有几种，请分别给出每一种制约关系在现实生活中的具体缘由和例子。

3.21 引入线程的根本原因是什么？

3.22 从调度、资源分配对进程和线程进行比较。

3.23 用户级线程与内核级线程最主要的区别是什么？

阅读材料

Process

A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

A computer program is a passive collection of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, openning up several instances of the same program often means more than one process is being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow fast response for interactive user applications. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This seeming execution of multiple processes simultaneously is called concurrency.

For security and reliability, most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

Representation

In general, a computer system process consists of (or is said to own) the following resources:

(1) An image of the executable machine code associated with a program.

(2) Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.

(3) Operating system descriptors of resources that are allocated to the process, such as file descriptors (UNIX terminology) or handles (Windows), and data sources and sinks.

(4) Security attributes, such as the process owner and the process' set of permissions

(allowable operations).

(5) Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

The operating system holds most of this information about active processes in data structures called process control blocks. Any subset of the resources, typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or child (daughter) processes.

The operating system keeps its processes separate and allocate the resources they need, so that they are less likely to interfere with each other and cause system failures (e. g. , deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Multitasking and process management

A multitasking operating system may just switch between processes to give the appearance of many processes executing simultaneously (that is, in parallel), though in fact only one process can be executing at any one time on a single-core CPU (unless using multithreading or other similar technology).

It is usual to associate a single process with a main program, and child processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to own resources, of which an image of its program (in memory) is one such resource. However, in multiprocessing systems many processes may run off of, or share, the same reentrant program at the same location in memory, but each process is said to own its own image of the program.

Processes are often called "tasks" in embedded operating systems. The sense of "process" (or task) is "something that takes up time", as opposed to "memory", which is "something that takes up space".

The above description applies to both processes managed by an operating system, and processes as defined by process calculi.

If a process requests something for which it must wait, it will be blocked. When the process is in the blocked state, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where regions of a process's memory may be really on disk and not in main memory at any time. Note that even unused portions of active processes/tasks (executing programs) are eligible for swapping to disk. All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.

Process states

The various process states, displayed in a state diagram, with arrows indicating possible transitions between states.

An operating system kernel that allows multitasking needs processes to have certain states. Names for these states are not standardized, but they have similar functionality.

First, the process is "created" by being loaded from a secondary storage device (hard disk drive, CD-ROM, etc.) into main memory. After that the process scheduler assigns it the "waiting" state.

While the process is "waiting", it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.

If a process needs to wait for a resource (wait for user input or file to open, for example), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.

Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory.

Inter-process communication

When processes communicate with each other it is called "Inter-process communication" (IPC). Processes frequently need to communicate, for instance in a shell pipeline, the output of the first process need to pass to the second one, and so on to the other process. It is preferred in a well-structured way not using interrupts.

It is even possible for the two processes to be running on different machines. The operating system (OS) may differ from one process to the other, therefore some mediator(s) (called protocols) are needed.

词汇表

1. multitasking n. 多任务
2. time-sharing system 分时系统
3. concurrency n. 并发
4. heap n. 堆
5. thrashing n. 抖动
6. spin-off 分拆
7. process calculi 进程演算