
# 进程和处理器管理

# 第 3 章 进程和处理器管理

为了描述程序在并发执行时对系统资源的共享，我们需要一个描述程序执行时动态特征的概念，这就是进程。在进程的基础上，引入线程的概念可进一步提高进程的并发性。处理器管理的工作是对处理器资源进行合理的分配使用，以提高处理器利用率，并使各用户公平地得到处理器资源。在本章中，我们将讨论进程和线程的概念、控制和相互关系，以及处理器调度算法。

## 3.1 进程

进程是处理器管理中的基本概念。本节讨论进程的概念、进程的状态和状态转换。

### 3.1.1 程序的顺序执行和并发执行

程序的执行可分为顺序执行和并发执行两种方式。顺序执行是指操作系统依次执行各程序，在一个程序的整个执行过程中该程序执行占用所有系统资源，不会中途暂停。顺序执行是单道批处理系统的执行方式，也用于简单的单片机系统。并发执行是指多个程序在一个处理器上的交替执行，这种交替执行在宏观上表现为同时执行。现代操作系统大多采用并发执行方式，具有许多新的特征。引入并发执行的目的是为了提高计算机资源的利用率。

程序的功能是依据指令对输入信息进行处理。程序的顺序执行具有下列三个特征：① 顺序性：程序的执行是按照程序结构所指定的次序进行的，可能的次序有分支、循环或跳转等；② 封闭性：程序在执行过程中独占全部资源，计算机的状态完全由该程序的控制逻辑所决定；③ 可再现性：只要程序执行的初始条件相同，执行结果就完全相同。例如，在程序中可利用空指令控制时间关系。

程序的并发执行可提高计算机资源的利用率，但并发执行也改变了程序的执行环境，并导致一些在顺序执行方式下可正常工作的程序在并发执行方式下不能正常工作。这种程序执行环境的变化体现在以下三个方面：① 间断(异步)性：处理器交替执行多个程序，每个程序都是以“走走停停”的方式执行，可能走到中途停下来，而且程序无法预知每次执行和暂停的时间长度，从而失去了原有的时序关系；② 失去封闭性：由于多个程序共享一个计算机系统的多种资源，因此每个程序的执行都会受其他程序的控制逻辑的影响，例如，一个程序写到存储器中的数据可能被另一个程序修改，失去原有的数据不变特征；③ 失去可再现性：由于程序执行环境的封闭性不再成立，因此程序每次执行的环境可能会不同，执行环境在程序的两次执行期间发生变化导致执行结果的不同，程序的执行结果失去原有的可重复特征。

程序执行是为了对输入信息进行处理，并得到相应的处理结果。为此，程序在并发执行时必须保持封闭性和可再现性。由于并发执行失去封闭性的原因是共享资源的影响，因此现在的工作就是去掉这种影响。

1966年，Bernstein给出了程序并发执行的条件。注意：他所给出的条件并没有考虑执行速度的影响。假设程序P(i)所访问的共享变量的读集和写集分别为R(i)和W(i)，则任意两个程序P(i)和P(j)可以并发执行的条件有以下三条：

1) $R(i) \cap W(j) = \Phi$。

2) $W(i) \cap R(j) = \Phi$。

3) $W(i) \cap W(j) = \Phi$。

其中，前两个条件保证一个程序在两次读操作之间存储器中的数据不会发生变化；最后一个条件保证程序的写操作的结果不会丢失。只要同时满足三个条件，并发执行的程序就可保持封闭性和可再现性。但这并没有解决所有问题，在实际的程序执行过程中很难对这三个条件进行检查。下面我们讨论如何通过操作系统的有效管理来保证并发执行程序的封闭性和可再现性。

### 3.1.2　进程的定义和描述

进程(process)是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。进程与处理器、存储器和外设等资源的分配和回收相对应，进程是计算机系统资源的使用主体。在操作系统中引入进程的并发执行，是指多个进程在同一计算机操作系统中的并发执行。引入进程并发执行可提高对硬件资源的利用率，但又带来额外的空间和时间开销，增加了操作系统的复杂性。

作为描述程序执行过程的概念，进程具有动态性、独立性、并发性和结构化等特征。动态性是指进程具有动态的地址空间，地址空间的大小和内容都是动态变化的。地址空间的内容包括代码（指令执行和处理器状态的改变）、数据（变量的生成和赋值）和系统控制信息（进程控制块的生成和删除）。独立性是指各进程的地址空间相互独立，除非采用进程间通信手段，否则不能相互影响。并发性也称为异步性，是指从宏观上看，各进程是同时独立运行的。结构化是指进程地址空间的结构划分，如代码段、数据段和核心段划分。

进程和程序是两个密切相关的不同概念，它们在以下几个方面存在区别和联系。

- 进程是动态的，程序是静态的。程序是有序代码的集合；进程是程序的执行。进程通常不可以在计算机之间迁移；而程序通常对应着文件、静态和可以复制。
- 进程是暂时的，程序是永久的。进程是一个状态变化的过程；程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序是密切相关的。通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。进程可创建其他进程，而程序并不能形成新的程序。

进程是程序代码的执行过程，但并不是所有代码执行过程都从属于某个进程。例如，处理器调度器是操作系统中的一段代码，它完成的功能包括：① 把处理器从一个进程切换到另一个进程；② 防止某进程独占处理器。处理器调度器的执行过程就不与进程相对应。

进程控制块（Process Control Block，PCB）是由操作系统维护的用来记录进程相关信息的数据结构。每个进程在操作系统中都有对应的进程控制块，操作系统维护的进程控制块总数可能会有所限制。操作系统依据进程控制块对进程进行控制和管理，进程控制块中的内容会随进程推进而动态改变。进程控制块处于操作系统核心，通常不能由应用程序自身的代码来直接访问，而要通过系统调用进行访问。在UNIX中也可通过进程文件系统(/proc)直接访问进程映像。

进程控制块的内容可分成进程描述信息、进程控制信息、资源占用信息和处理器现场保护结构这4个部分。进程描述信息包括进程标识符 (process ID)、进程名（通常是可执行文件名）、用户标识符(user ID)和进程组(process group)等。进程控制信息包括当前状态、优先级、代码执行入口地址、程序的外存地址、运行统计信息（执行时间、页面调度）、进程阻塞原因等。资源占用信息是指进程占用的系统资源列表。处理器现场保护结构保存寄存器值，如通用寄存器、程序计数器PC、状态字PSW，地址包括栈指针等。

操作系统要将处于同一状态的进程的进程控制块组织在一起，常用的组织方式有链表和索引表两种。链表方式是将同一状态的进程控制块组成一个链表，多个状态对应多个不同的链表，如就绪链表和阻塞链表等。索引表方式是将同一状态的进程归入一个索引表，再由索引指向相应的进程控制块，多个状态对应多个不同的索引表，如就绪索引表和阻塞索引表等。

对进程执行活动全过程的静态描述称为进程上下文。进程上下文包括进程的用户地址空间内容、处理器中寄存器内容及与该进程相关的核心数据结构等，可分成用户级上下文、寄存器级上下文和系统级上下文。用户级上下文是指进程的用户地址空间，包括用户正文段、用户数据段和用户栈。寄存器级上下文是指程序寄存器、处理器状态寄存器、栈指针、通用寄存器的值等。系统级上下文包括进程的静态部分（PCB和资源表格）和由核心栈等构成的动态部分。

### 3.1.3 进程的状态转换

进程在从创建到终止的全过程中一直处于一个不断变化的过程。为了刻画进程的这个变化过程，所有操作系统都把进程分成若干种状态，约定各种状态间的转换条件。对进程状态的刻画也经历了一个不断精确化的过程。下面我们就讨论进程的状态模型。

1. 五状态进程模型

在五状态进程模型中，进程状态被分成下列五种状态。进程在运行过程中主要是在就绪、运行和阻塞三种状态间进行转换。创建状态和退出状态描述进程创建的过程和进程退出的过程。如图3-1所示。

1) 运行状态(Running)：进程占用处理器资源；处于此状态的进程的数目小于等于处理器的数目。在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的空闲进程。

2) 就绪状态(Ready)：进程已获得除处理器外的所需资源，等待分配处理器资源；只要分配了处理器进程就可执行。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由于 I/O操作完成而进入就绪状态时，排入高优先级队列。

3) 阻塞状态(Blocked)：当进程由于等待I/O操作或进程同步等条件而暂停运行时，它处于阻

塞状态。在条件满足之前，即使把处理器分配给该进程，它也是无法继续执行的。

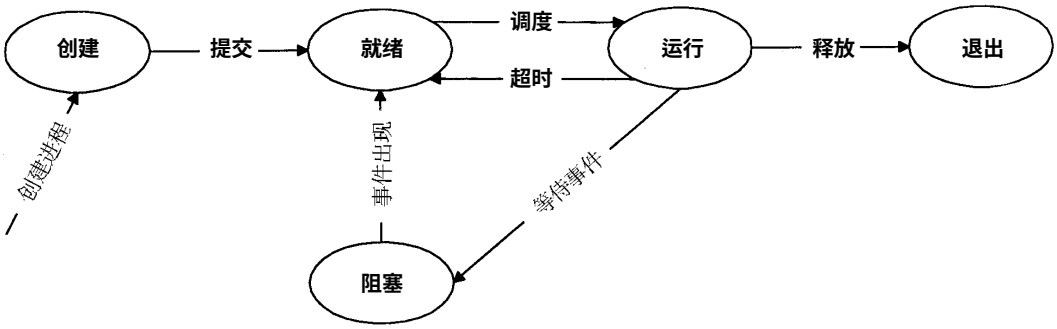


图3-1　五状态进程模型(状态变迁)

4) 创建状态(New)：进程正在创建过程中，还不能运行。操作系统在创建状态要进行的工作包括分配和建立进程控制块表项、建立资源表格（如打开文件表）并分配资源、加载程序并建立地址空间表等。

5) 退出状态(Exit)：进程已结束运行，回收除进程控制块之外的其他资源，并让其他进程从进程控制块中收集有关信息（如记帐和将退出代码传递给父进程）。

五状态进程模型中的状态转换主要包括下列几种。操作系统中多个进程的并发执行是通过调度与超时两种转换间的循环，或调度、等待事件和事件出现三种转换间的循环来描述的。

1) 创建新进程：创建一个新进程，以运行一个程序。创建新进程的可能原因包括用户登录、操作系统创建以提供某项服务、批处理作业等。

2) 收容(Admit, 也称为提交)：收容一个新进程，进入就绪状态。由于性能、内存、进程总数等原因，系统会限制并发进程总数。

3) 调度运行(Dispatch)：从就绪进程表中选择一个进程，进入运行状态。

4) 释放(Release)：由于进程完成或失败而终止进程运行，进入结束状态。为了简洁，状态变迁图中只画出了运行状态到退出状态间的释放转换；但实际上，还存在从就绪状态或阻塞状态到退出状态的释放转换。运行到结束的转换可分为正常退出 (Exit)和异常退出(abort)；其中异常退出是指进程执行超时、内存不够、非法指令或地址访问、 I/O操作失败、被其他进程所终止等原因而退出。从就绪状态或阻塞状态到结束状态的释放转换可能是由于多种原因引发，如父进程可在任何时间终止子进程。

5) 超时（Timeout）：由于用完时间片或高优先级进程就绪等原因导致进程暂停运行。

6) 事件等待（Event Wait）：进程要求的事件未出现而进入阻塞；可能的原因包括：申请系统服务或资源、通信、 I/O操作等。

7) 事件出现（Event Occurs）：进程等待的事件出现；如操作完成、申请成功等。

对于五状态进程模型，操作系统要解决的一个重要问题是当一个事件出现时如何检查阻塞进程表中的进程状态，如图 3-2所示。当进程多时，这对系统性能影响很大。如图 3-3所示，一种可能的做法是按等待事件类型，排成多个队列。

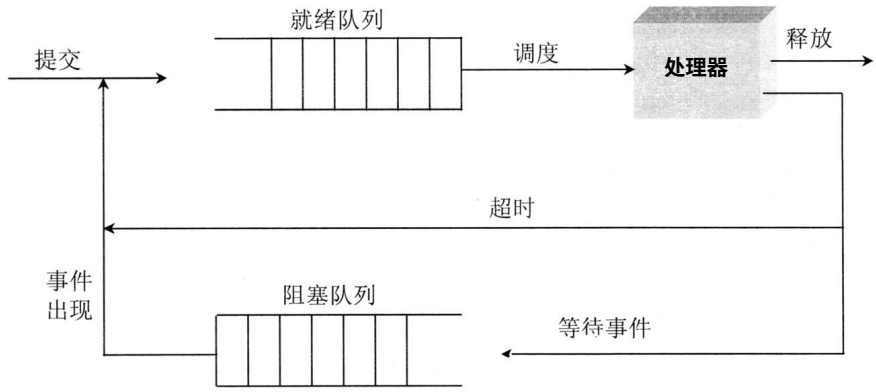


图3-2　五状态进程模型(单队列结构)

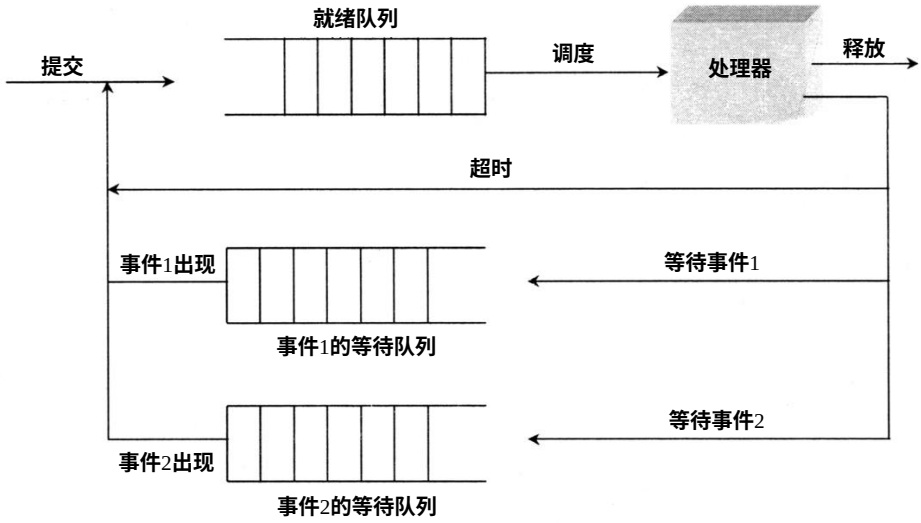


图3-3　五状态进程模型(多队列结构)

2. 挂起进程模型

五状态进程模型没有区分进程地址空间位于内存还是外存，而在操作系统中引入虚拟存储管理技术后，需要进一步区分进程的地址空间状态。这个问题的出现是由于进程优先级的引入，一些低优先级进程可能等待较长时间，从而被对换至外存。这种做法可得到下列的好处：① 提高处理器效率：就绪进程表为空时，有空闲内存空间用于提交新进程，可提高处理器效率；② 可为运行进程提供足够内存：资源紧张时，可把某些进程对换至外存；③ 有利于调试：在调试时，挂起被调试进程，可方便对其地址空间进行读写。

与五状态进程模型相比，挂起进程模型把原来的就绪状态和阻塞状态进行了细分。在单挂起进程模型中增加了一个阻塞挂起状态（见图 3-4）；而在双挂起进程模型中增加了就绪挂起和阻塞挂起两个状态（见图3-5)。这时，原来的就绪状态和阻塞状态的意义也发生了一些变化。下面

列出的是在挂起进程模型中四种意义有变化的状态或新的状态。

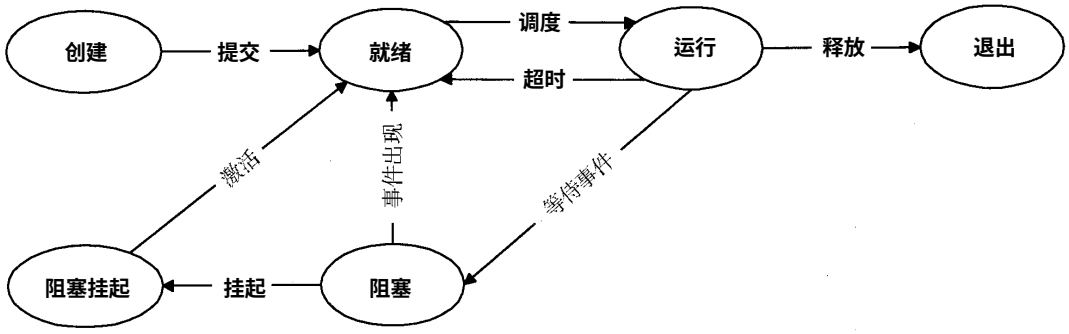


图3-4　单挂起进程模型

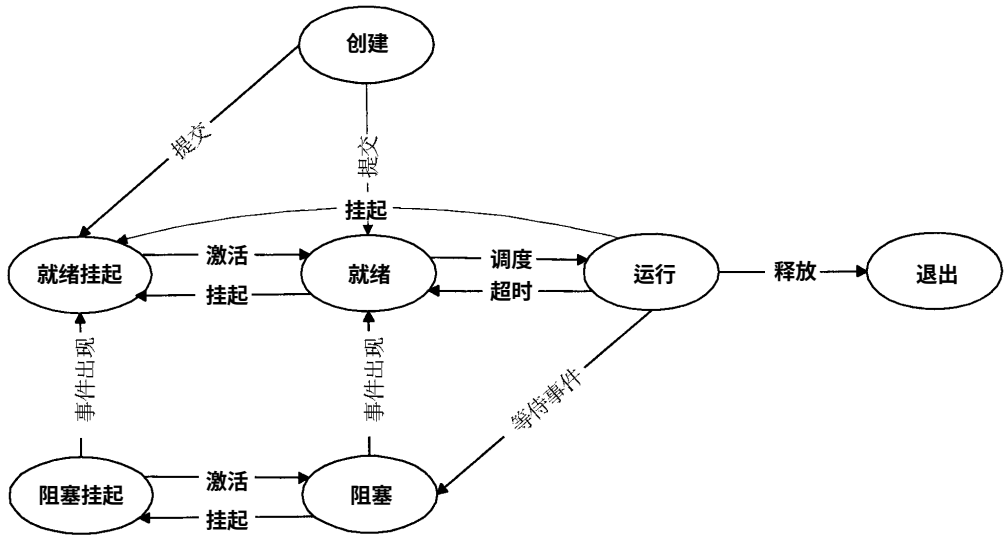


图3-5　双挂起进程模型

1) 就绪状态(ready)：进程在内存且可立即进入运行状态。

2) 阻塞状态(blocked)：进程在内存并等待某事件的出现。

3) 阻塞挂起状态（blocked, suspend)：进程在外存并等待某事件的出现。

4) 就绪挂起状态（ready, suspend)：进程在外存，但只要进入内存，即可运行。

在挂起进程模型中，新引入的状态转换有挂起和激活两类，意义有变化的状态转换有事件出现和收容两类。

1) 挂起（suspend)：把一个进程从内存转到外存；可能有以下几种情况：

- 阻塞到阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，会进行这种转换，以提交新进程或运行就绪进程。
- 就绪到就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先级就绪进

程时，系统会选择挂起低优先级就绪进程。

- 运行到就绪挂起：对抢先式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态。

2) 激活（activate）：把一个进程从外存转到内存，可能有以下几种情况：

- 就绪挂起到就绪：就绪挂起进程优先级高于就绪进程或没有就绪进程时，会进行这种转换。
- 阻塞挂起到阻塞：当一个进程释放足够内存时，系统会把一个高优先级阻塞挂起进程激活，系统认为会很快出现该进程所等待的事件。

3) 事件出现（event occur）：进程等待的事件出现，如操作完成、申请成功等；可能的情况有：

- 阻塞到就绪：针对内存进程的事件出现。
- 阻塞挂起到就绪挂起：针对外存进程的事件出现。

4) 收容(admit)：收容一个新进程，进入就绪状态或就绪挂起状态。进入就绪挂起的原因是系统希望保持一个大的就绪进程表（挂起和非挂起）。

我们这里讨论的进程状态模型是对实际操作系统中所使用的进程状态定义的抽象和简化，我们将在操作系统实例中介绍它们与实际操作系统中的进程状态之间的关系。

## 3.2 进程控制

操作系统对进程的控制是依据用户命令和系统状态来决定的。进程控制的功能是完成进程状态的转换。用户可在一定程序上对进程的状态进行控制。这种控制主要体现在进程的创建与退出，以及进程的挂起与激活。

### 3.2.1 进程的创建和退出

在操作系统完成初始化后，系统就可创建进程。在进程的创建过程中，操作系统要进行进程控制块等相关数据结构的维护。一个进程可利用系统调用功能来创建新的进程，创建者称为父进程，而被创建的新进程称为子进程。按子进程是否覆盖父进程和是否加载新程序，子进程的创建可分为如表3-1中所示的fork、spawn和exec三种类型。由于复制现有进程的上下文时一定会产生新进程，所以只有三种类型。

表 3-1

| | 产生新进程 | 不产生新进程 |
|---|---|---|
| 复制现有进程的上下文 | fork（新进程的系统上下文会有不同） | — |
| 加载程序 | spawn（创建新进程并加载新程序） | exec（加载新程序并覆盖自身） |

子进程与父进程存在密切的关系，子进程的许多属性就是从父进程继承来的；与此同时，子进程又与父进程有区别，形成自己独立的属性。子进程可以从父进程中继承的属性包括：用户标识符、环境变量、打开文件、文件系统的当前目录、控制终端、已经连接的共享存储区和信号处

理例程入口表等。子进程不能从父进程继承的属性包括：进程标识符和父进程标识符等。还有其他一些属性可在进程创建中约定是否能从父进程继承。各种操作系统都有相应的系统调用完成进程创建，主要工作过程都是一致的，但它们也会在细节上有一些区别。

进程的退出是通过相应的系统调用进行的，也称为“进程终止”。例如，在C语言中可调用exit()来终止进程。在进程的退出过程中，操作系统要删除系统维护的相关数据结构并回收进程占用的系统资源，例如，释放进程占用的内外存空间、关闭所有打开文件、释放共享内存段和解除各种锁定等。

### 3.2.2　进程的阻塞和唤醒

进程在执行过程中会因为等待I/O操作完成或等待某个事件出现而进入阻塞状态。当处于阻塞状态的进程所等待的操作完成或事件出现时，进程将会从阻塞状态唤醒而进入就绪状态。用户可通过相应系统调用来等待某个事件或唤醒某个阻塞进程。

UNIX系统中，与进入阻塞状态相关的系统调用主要有：暂停一段时间(sleep)、暂停并等待信号(pause)和等待子进程暂停或终止(wait)。与唤醒阻塞进程相关的系统调用主要是发送信号到某个或一组进程(kill)。各系统调用的简要描述如下：

1) sleep将在指定的时间seconds内挂起本进程。其调用格式为：“unsigned sleep(unsigned seconds);”，返回值为实际的挂起时间。

2) pause挂起本进程以等待信号，接收到信号后恢复执行。当接收到终止进程信号时，该调用不再返回。其调用格式为“int pause(void);”。

3) wait挂起本进程以等待子进程的结束，子进程结束时返回。当父进程创建多个子进程且已有子进程退出时，父进程中wait函数在第一个子进程结束时返回。其调用格式为“pid_t wait(int *stat_loc);”，返回值为子进程ID。

4) kill可发送信号sig到某个或一组进程pid。其调用格式为：“int kill(pid_t pid, int sig);”。信号的定义在文件“/usr/ucbinclude/sys/signal.h”中。命令“kill”可用于向进程发送信号。例如，“kill -9100”将发送SIGKILL到ID为100的进程；该命令将终止该进程的执行。

在Windows NT和Windows 2000/XP中，处理器的调度对象为线程，用户可通过系统调用SuspendThread和ResumeThread来挂起或激活线程。一个线程可被多次挂起和多次激活。在线程控制块中有一个挂起计数（suspend count），挂起操作使该计数加1，激活操作使该计数减1。当挂起计数从0变为1时，线程进入阻塞状态；当挂起计数由1变为0时，线程恢复执行。各系统调用的简要描述如下：

1) 挂起：Windows NT中的SuspendThread可挂起指定的线程。

```
DWORD SuspendThread( HANDLE hThread线程句柄
);
```

2) 激活：Windows NT中的ResumeThread可恢复指定线程的执行。

```
DWORD ResumeThread( HANDLE hThread表示被恢复的线程
);
```

### 3.2.3 Windows 2000/XP进程管理

Windows 2000/XP中的进程是系统资源分配的基本单位。Windows 2000/XP进程是作为对象来管理的，可通过相应句柄 (handle)来引用进程对象，操作系统提供一组控制进程对象的服务(services)。进程对象的属性包括：进程标识 (PID)、资源访问令牌(Access Token)、进程的基本优先级(Base Priority)和默认亲合处理器集合(Processor Affinity)等。

为了支持Win32、OS/2、POSIX等多种运行环境子系统，Windows 2000/XP核心的进程之间没有任何关系（包括父子关系），各运行环境子系统分别建立、维护和表达各自的进程关系。例如，POSIX环境子系统维护 POSIX应用进程间的父子关系。如图 3-6所示，Windows NT和Windows 2000/XP把Win32环境子系统设计成整个系统的主子系统，一些基本的进程管理功能被放置在Win32子系统中，POSIX和OS/2等其他的子系统会利用 Win32子系统的功能来实现自身的功能。在Windows 2000/XP中，与一个运行环境子系统中的应用进程相关的进程控制块信息会分布在本运行环境子系统、Win32子系统和系统内核中。

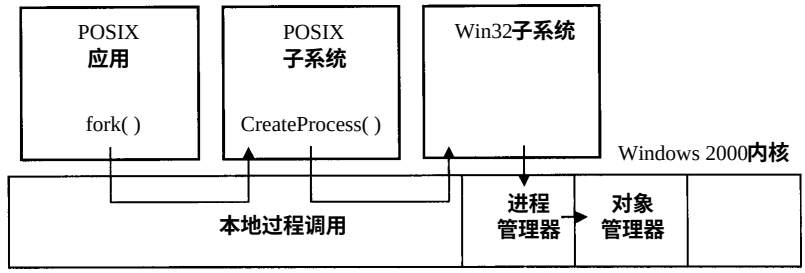


图3-6 Windows 2000/XP的进程关系

如图3-7所示，Windows 2000/XP中的每个Win32进程都由一个执行体进程块(EPROCESS)表示，执行体进程块描述进程的基本信息，并指向其他与进程控制相关的数据结构。执行体进程块中的主要内容包括：① 线程块列表：描述属于该进程的所有线程的相关信息，以便线程调度器进行处理器资源的分配和回收；② 虚拟地址空间描述表(Virtual Address space Descriptor, VAD)：描述进程地址空间各部分属性，用于虚拟存储管理；③ 对象句柄列表：当进程创建或打开一个对象时，就会得到一个代表该对象的句柄，用于对象访问，对象句柄列表维护该进程正在访问的所有对象列表。

Windows 2000/XP支持的各环境子系统都有相应的系统调用实现进程控制。Win32子系统的进程控制系统调用主要有CreateProcess、ExitProcess和TerminateProcess，系统调用CreateProcess用于进程创建，而ExitProcess和TerminateProcess用于进程退出。这几个系统调用的简要介绍如下：

1) CreateProcess创建新进程及其主线程，以执行指定的程序。Win32进程在创建时可指定从父进程继承的属性，许多对象句柄的继承特征可在对象创建或打开时指定，从而影响新进程的执行。新进程可以继承的进程属性包括：打开文件的句柄、各种对象（如进程、线程、信号量、管道等）的句柄、环境变量、当前目录、原进程的控制台、原进程的进程组标识符等。新进程不能从父进程继承的属性包括：优先权类、内存句柄、DLL模块句柄等。

2) ExitProcess和TerminateProcess都可用于进程退出，它们会终止调用进程内的所有线程。

这两个系统调用的区别在于终止操作是否完整。 ExitProcess终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、所有线程等。 TerminateProcess终止指定的进程和它的所有线程；它的终止操作是不完整的，通常只用于异常情况下对进程的终止。

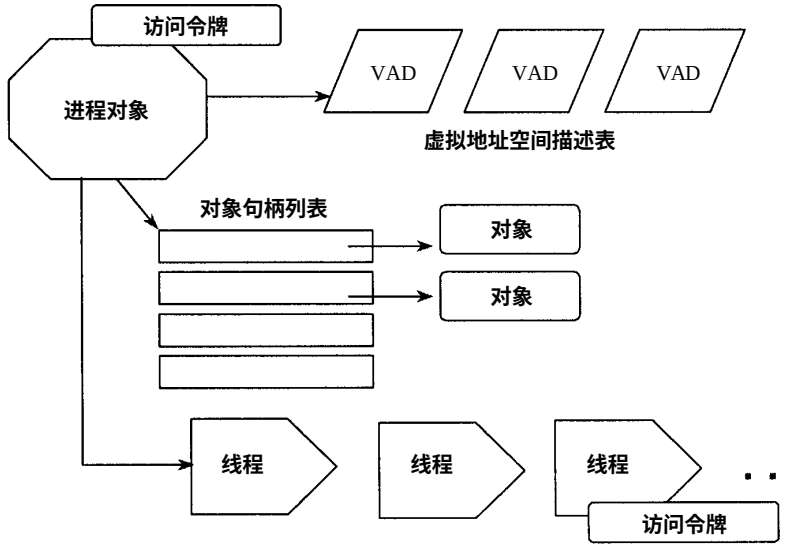


图3-7　Windows 2000/XP中的Win32进程结构

Windows 2000/XP提供一整套机制用于程序调试。在操作系统维护的进程对象属性中包括一个用于调试时进程间通信的通道，通过此通道可了解和控制被调试进程的运行，访问被调试进程地址空间的内容。调试器进程（debugger）在创建被调试进程（target）时可指定DEBUG_ PROCESS标志或利用DebugActiveProcess函数，在调试器与被调试进程间建立调试关系，被调试进程会向调试器通报所有调试事件。被调试进程向调试器进程发送的调试事件包括：创建新进程、新线程、加载DLL、执行断点等。调试器可通过WaitForDebugEvent和ContinueDebugEvent来等待被调试进程的调试事件和继续调试进程的运行。 WaitForDebugEvent可在指定的时间内等待可能的调试事件；ContinueDebugEvent可使被调试事件暂停的进程继续运行。调试器进程可通过ReadProcessMemory()和WriteProcessMemory()来读写被调试进程的存储空间。

## 3.3　线程

在操作系统中，进程的引入提高了计算机资源的利用效率。但在进一步提高进程的并发性时，人们发现进程切换开销占的比重越来越大，同时进程间通信的效率也受到限制。线程的引入正是为了简化线程间的通信，以小的开销来提高进程内的并发程度。本节讨论线程的概念和它与进程的差异。

### 3.3.1　线程的概念

在只有进程概念的操作系统中，进程是存储器、外设等资源的分配单位，同时也是处理器调

度的对象。为了提高进程内的并发性，在引入线程的操作系统中，把线程作为处理器调度的对象，而把进程作为资源分配单位，一个进程内可同时有多个并发执行的线程。

线程(Thread)是一个动态的对象，它是处理器调度的基本单位，表示进程中的一个控制点，执行一系列的指令。由于同一进程内各线程都可访问整个进程的所有资源，因此它们之间的通信比进程间通信要方便；而同一进程内的线程间切换也会由于许多上下文的相同而简化。如图 3-8所示，线程与进程是两个密切相关的概念。我们可以把原来的进程概念理解为只有一个主线程的进程。

同一进程内各线程的差异主要体现在线程状态、寄存器上下文和堆栈等必不可少的线程执行环境上。这样，在操作系统中引入线程概念，就可减小并发执行的时间和空间开销，容许通过在系统中建立更多的线程来提高并发性。线程的优点具体体现在以下几方面：① 线程的创建时间比进程短；② 线程的终止时间比进程短；③ 同进程内的线程切换时间比进程短；④ 由于同进程内线程间共享内存和文件资源，因此可直接进行不通过内核的通信。

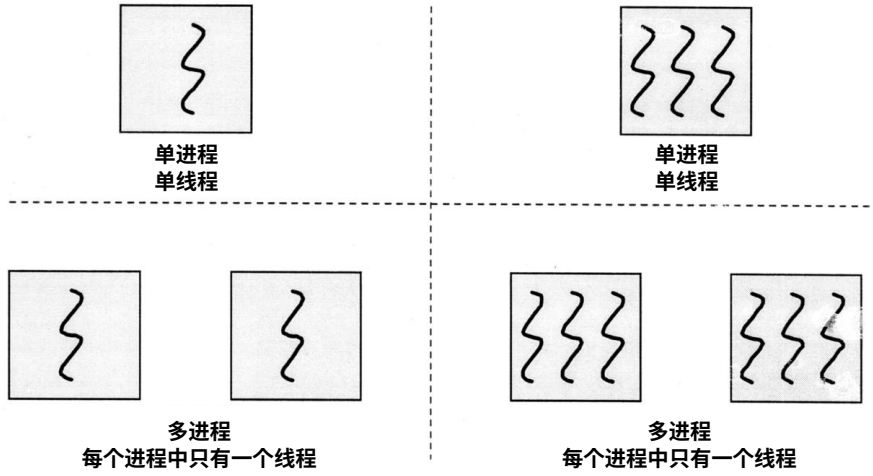


图3-8 进程与线程的关系

在操作系统中有多种方式可实现对线程的支持。最自然的方法是由操作系统内核提供线程的控制机制。在只有进程概念的操作系统中可由用户程序利用函数库提供线程的控制机制。还有一种做法是同时在操作系统内核和用户程序两个层次上提供线程控制机制。这就构成了内核线程、用户线程和轻量级进程这三种线程的实现方式。

内核线程(kernel-level thread)是指由操作系统内核完成创建和撤销，用来执行一个指定的函数线程。在支持内核线程的操作系统中，内核维护进程和线程的上下文信息以及线程切换由内核完成。一个内核线程由于I/O操作而阻塞，不会影响其他线程的运行。这时处理器时间片分配的对象是线程，所以多线程的进程获得更多处理器时间。 Windows NT和Windows 2000/XP支持内核线程。

用户线程(user-level thread)是指不依赖于操作系统核心，由应用进程利用线程库提供创建、

同步、调度和管理线程的函数来控制的线程。由于用户线程的维护由应用进程完成，不需要操作系统内核了解用户线程的存在，因此可用于不支持内核线程的多进程操作系统，甚至是单用户操作系统。用户线程切换不需要内核特权，用户线程调度算法可针对应用优化。在许多应用软件中都有自己的用户线程。例如数据库系统Informix和图形处理软件Aldus PageMaker等。由于用户线程的调度在应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态 /核心态切换，因此速度特别快。当然，由于操作系统内核不了解用户线程的存在，当一个线程在进入系统调用后阻塞时，整个进程都必须等待。这时处理器时间片是分配给进程的，进程内有多个线程时，每个线程的执行时间相对就少。

轻量级进程(LightWeight Process)是指由内核支持的用户线程。一个进程可有一个或多个轻量级进程，每个轻量级进程由一个单独的内核线程来支持。由于同时提供内核线程控制机制和用户线程库，因此可很好地把内核线程和用户线程的优点结合起来。

### 3.3.2　进程和线程的比较

由于进程与线程的密切相关，因此我们有必要比较一下进程与线程的差异（见图 3-9）。可从以下三个角度来比较进程和线程的不同。① 地址空间资源：不同进程的地址空间是相互独立的，而同一进程的各线程共享同一地址空间。一个进程中的线程在另一个进程中是不可见的。② 通信关系：进程间通信必须使用操作系统提供的进程间通信机制，而同一进程中的各线程间可以通过直接读写进程数据段（如全局变量）来进行通信。当然同一进程中各线程间的通信也需要同步和互斥手段的辅助，以保证数据的一致性。③ 调度切换：同一进程中的线程上下文切换比进程上下文切换要快得多。

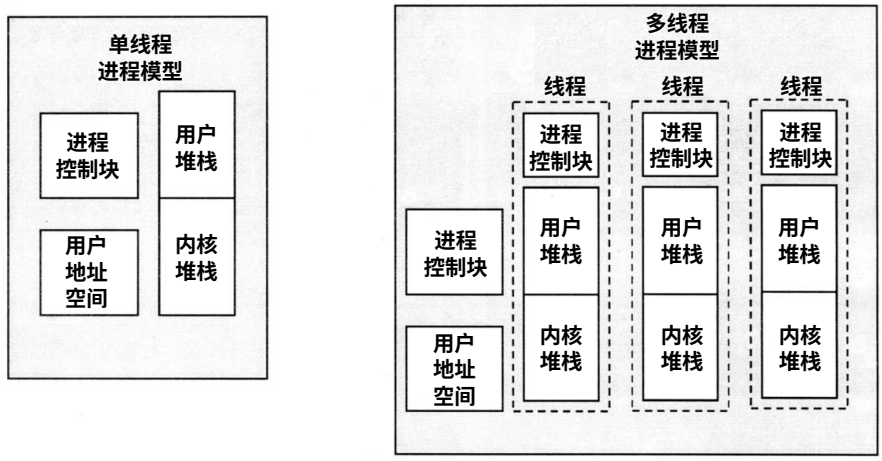


图3-9　进程和线程的比较

### 3.3.3　Windows 2000/XP线程

Windows 2000/XP的线程是内核线程，系统的处理器调度对象为线程。线程上下文主要包括

寄存器、线程环境块、核心栈和用户栈。 Windows 2000/XP把线程状态分成下面七种，如图 3-10 所示，与单挂起进程模型很相似，它们的主要区别在于从就绪状态到运行状态的转换中间多了一个备用状态，以优化线程的抢先特征。

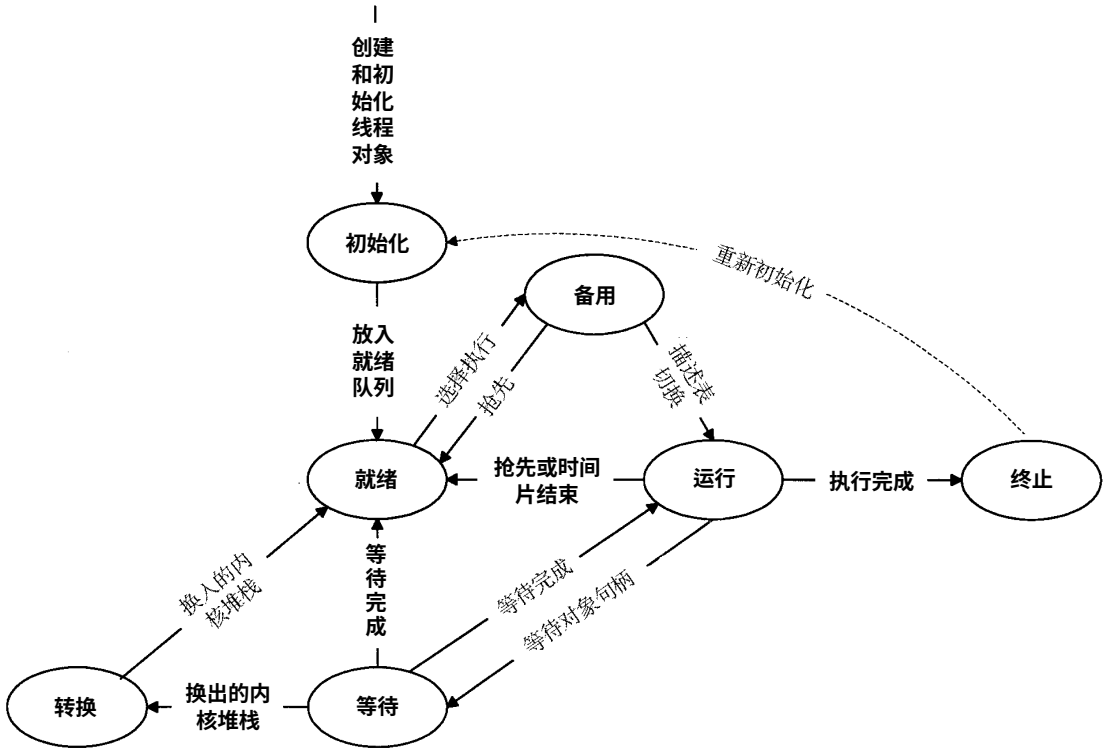


图3-10 Windows 2000/XP的线程状态

1) 就绪状态(ready)：线程已获得除处理器外的所需资源，正等待调度执行。

2) 备用状态(standby)：已选择好线程的执行处理器，正等待描述表切换，以进入运行状态。系统中每个处理器上只能有一个处于备用状态的线程。

3) 运行状态(running)：已完成描述表切换，线程进入运行状态。线程会一直处于运行状态，直到被抢先、时间片用完、线程终止或进入等待状态。

4) 等待状态(waiting)：线程正等待某对象，以同步线程的执行。当等待事件出现时，等待结束，并根据优先级进入运行或就绪状态。

5) 转换状态(transition)：转换状态与就绪状态类似，但线程的内核堆栈位于外存。当线程等待事件出现而它的内核堆栈处于外存时，线程进入转换状态；当线程内核堆栈被调回内存时，线程进入就绪状态。

6) 终止状态(terminated)：线程执行完就进入终止状态；如执行体有一个指向线程对象的指针，可将处于终止状态的线程对象重新初始化，并再次使用。

7) 初始化状态(Initialized)：线程创建过程中的线程状态。

Windows 2000/XP有一组相关的系统调用用于线程控制。 CreateThread完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。 ExitThread用于结束当前线程。 SuspendThread可挂起指定的线程。 ResumeThread可激活指定线程，它的对应操作是递减指定线程的挂起计数，当挂起计数减为 0时，线程恢复执行。

## 3.4　进程互斥和同步

由于多进程在操作系统中的并发执行，它们之间存在着相互制约的关系。这就是进程间的同步和互斥关系。进程同步是指多个进程中发生的事件存在某种时序关系，必须协同动作、相互配合，以共同完成一个任务。进程互斥是指由于共享资源所要求的排它性，进程间要相互竞争，以使用这些互斥资源。下面讨论如何实现进程同步和互斥。

### 3.4.1　互斥算法

进程互斥的解决有两种做法，一是由竞争各方平等协商；二是引入进程管理者，由管理者来协调竞争各方对互斥资源的使用。这两种做法在操作系统中都存在，我们首先讨论第一种做法，即基于进程间平等协商的互斥算法。

从多道系统开始，程序的运行环境就存在资源共享问题。多个进程之间需要对有限的资源进行共享，各进程在运行时是否能得到所需要的资源，是受其他进程的影响的。如果一个进程所需要的资源已被其他进程占用，该进程就无法正常运行下去。临界资源是指计算机系统中的需要互斥使用的硬件或软件资源，如外设、共享代码段、共享数据结构等。多个进程在对临界资源进行访问时，特别是进行写入或修改操作时，必须互斥地进行。计算机系统中也有一些可以同时访问的共享资源不是临界资源，如只读数据。

在多进程系统中，我们可把进程间的相互制约关系按相互感知程度分成下面表 3-2所列的三种类型。我们可以把计算机系统中资源共享的程度分成这样三个层次：互斥 (mutual exclusion)、死锁(deadlock)和饥饿(starvation)。保证资源的互斥使用是指多个进程不能同时使用同一个资源，这是正确使用资源的最基本要求。避免死锁是指避免多个进程互不相让，都得不到足够资源的情况出现，从而保证系统功能的正常运行。避免饥饿是指避免某些进程一直得不到资源或得到资源的概率很小，从而保障系统内资源使用的公平性。

表 3-2

| 相互感知的程度 | 交互关系 | 一个进程对其他进程的影响 | 潜在的控制问题 |
|---|---|---|---|
| 相互不感知(完全不了解其他进程的存在) | 竞争(competition) | 一个进程的操作对其他进程的结果无影响 | 互斥、死锁、饥饿 |
| 间接感知(双方都与第三方交互，如共享资源) | 通过共享进行协作 | 一个进程的结果依赖于从其他进程获得的信息 | 互斥、死锁、饥饿 |
| 直接感知(双方直接交互，如通信) | 通过通信进行协作 | 一个进程的结果依赖于从其他进程获得的信息 | 死锁、饥饿 |

为了保证临界资源的正确使用，我们可把临界资源的访问过程分成如图 3-11所示的四个部分。① 进入区(entry section)：为了进入临界区使用临界资源，在进入区要检查可否进入临界区；如果可以进入临界区，通常设置相应的“正在访问临界区”标志，以阻止其他进程同时进入临界区。② 临界区(critical section)：进程中访问临界资源的一段代码。③ 退出区(exit section)：将“正在访问临界区”标志清除。④ 剩余区(remainder section)：代码中的其余部分。

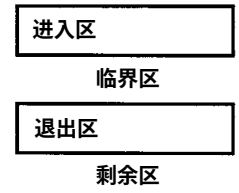


图3-11 临界区的访问过程

为了合理使用计算机系统中的资源，在操作系统中采用的进程同步机制应遵循以下几条准则。① 空闲则入：任何同步机制都必须保证任何时刻最多只有一个进程位于临界区；当有进程位于临界区时，任何其他进程均不能进入临界区。② 忙则等待：当已有进程处于其临界区时，后到达的进程只能在进入区等待。③ 有限等待：为了避免死锁等现象的出现，等待进入临界区的进程不能无限期地“死等”。④ 让权等待：因在进入区等待而不能进入临界区的进程，应释放处理器，转换到阻塞状态，以使其他进程有机会得到处理器的使用权。

1. 进程互斥的软件方法

通过平等协商方式实现进程互斥的最初方法是软件方法。其基本思路是在进入区检查和设置一些标志，如果已有进程在临界区，则在进入区通过循环检查进行等待；在退出区修改标志。其中的主要问题是设置什么标志和如何检查标志。下面我们讨论几种用软件方法实现的软件互斥算法。

(1) 算法1：单标志算法

假设有两个进程$P_i$和$P_j$。设立一个公用整型变量 turn，描述允许进入临界区的进程标识。每个进程都在进入区循环检查变量turn是否允许本进程进入。即 turn为i时，进程$P_i$可进入；否则循环检查该变量，直到 turn变为本进程标识。在退出区修改允许进入进程的标识。即进程 $P_i$退出时，改turn为进程$P_j$的标识j。图3-12所示为进程$P_i$的代码。

```
while (turn ! = i);
```
临界区
```
turn = j;
```
剩余区

图3-12 互斥算法1（单标志）

算法1可以保证任何时刻最多只有一个进程在临界区。但它的缺点是强制轮流进入临界区，没有考虑进程的实际需要。这种算法容易造成资源利用不充分。例如，在 $P_i$出让临界区之后，$P_j$使用临界区之前，$P_i$不可能再次使用临界区。

(2) 算法2：双标志、先检查算法

为了克服算法 1的缺点，我们考虑修改临界区标志的设置。设立一个标志数组 flag[ ]，描述各进程是否在临界区，初值均为 FALSE。在进入区的操作为：先检查，后修改。即在进入区先检查另一个进程是否在临界区，不在时修改本进程在临界区的标志，表示本进程在临界区。在退出区修改本进程在临界区的标志，表示本进程不在临界区。图 3-13所示为进程$P_i$的代码。

```
while (flag[j]);                <a>
flag[i] = TRUE;                 <b>
```
临界区
```
flag [i] = FALSE;
```
剩余区

图3-13 互斥算法2（双标志、先检查）

算法2的优点是克服了算法 1的缺点，两个进程不用交替进入，可连续使用。但由于使用多个

标志，算法又产生一个新问题，即进程 $P_i$和$P_j$可能同时进入临界区，从而违反了最多只有一个进程在临界区的要求。我们按序列“ $P_i$<a> $P_j$<a> $P_i$<b> $P_j$<b> ”执行时，就会出现进程$P_i$和$P_j$同时进入的问题。即进程在检查对方标志 flag之后和切换自己标志flag之前有一段时间间隔，这个时间间隔导致两个进程都在进入区通过检查。这个问题出在检查和修改操作不能连续进行。

(3) 算法3：双标志、先修改后检查算法

为了解决算法2的新问题；我们有两种选择：一是保证检查和修改操作之间不出现间隔，一是修改标志含义。第一种方法是仅使用软件方法无法做到的，我们采用第二种方法。算法 3类似于算法2，它们的区别在于进入区操作是先修改后检查。这时标志 flag[i]表示进程$P_i$想进入临界区，而不再表示进程 i在临界区。图3-14所示为进程Pi的代码。

```
flag[i] = TRUE;                    <b>
while (flag[j]);                   <a>
        临界区
flag[i] = FALSE;
        剩余区
```

图3-14　互斥算法3（双标志、先修改后检查）

算法3可防止两个进程同时进入临界区。但它的缺点是 $P_i$和$P_j$可能都进入不了临界区。我们按序列“ $P_i$<b> $P_j$<b> $P_i$<a> $P_j$<a> ”执行时，就会都进不了临界区。即在修改本进程标志 flag之后和检查对方flag之前有一段时间间隔，这个间隔导致两个进程都想进入临界区，从而在检查对方标志时不通过。

(4) 算法4：先修改、后检查、后修改者等待算法

算法4的基本思想是结合算法 1和算法3。标志flag[i]表示进程$P_i$想进入临界区，标志turn表示同时修改标志时要在进入区等待的进程标识。在进入区先修改后检查。通过修改同一标志 turn来描述标志修改的先后；检查对方标志 flag，如果对方不想进入临界区则自己进入；否则再检查标志turn，由于标志turn中保存的是较晚的一次赋值，因此较晚修改标志的进程等待，较早修改标志的进程进入临界区。图3-15所示为进程$P_i$的代码。

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
        临界区
flag[i] = FALSE;
        剩余区
```

图3-15　互斥算法4（先修改、后检查、后修改者等待）

至此，算法4可完全正常工作。即实现了同步机制要求的四条准则中的前两条：空闲则入和忙则等待。但我们从上面的软件实现方法中可发现，对于两个进程间的互斥和三个以上进程间的互斥的进入区是要区别对待的，而这里最主要的问题就是修改标志和检查标志不能作为一个整体不被执行。下面的硬件方法就是利用处理器的指令系统来解决这个问题。

2. 进程互斥的硬件方法

完全利用软件方法实现进程互斥有很大局限性，如不适用于数目很多的进程间的互斥。现在已很少单独采用软件方法。在平等协商时都利用某些硬件指令来实现进程互斥。硬件方法的主要思路是用一条指令完成读和写两个操作，因而保证读操作与写操作不被打断。依据所采用的指令的不同硬件方法分成TS指令和Swap指令两种。

(1) TS (Test-and-Set)指令

TS指令的功能是读出指定标志后把该标志设置为TRUE。TS指令的功能可描述成下面的函数。

```
boolean TS(boolean *lock) {
  boolean old;
  old = *lock;  *lock = TRUE;
  return old;
}
```

利用TS指令实现的进程互斥算法是，每个临界资源设置一个公共布尔变量 lock，表示资源的两种状态：TRUE表示正被占用，FALSE表示空闲，初值为FALSE。在进入区利用TS进行检查和修改标志lock；有进程在临界区时，重复检查，直到其他进程退出时，检查通过。如图 3-16所示，所有要访问临界资源的进程的进入区和退出区代码都是相同的。

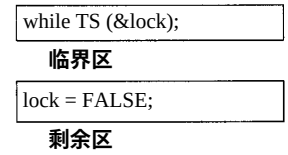


图3-16 互斥算法（TS指令）

(2) Swap指令（或Exchange指令）

Swap指令的功能是交换两个字（字节）的内容。我们可用下面的函数描述Swap指令的功能。

```
void SWAP(int *a, int *b) {
  int temp;
  temp = *a;  *a = *b;  *b = temp;
}
```

利用Swap指令实现的进程互斥算法是，每个临界资源设置一个公共布尔变量lock，初值为FALSE；每个进程设置一个私有布尔变量key，用于与lock间的信息交换。在进入区利用 Swap指令交换lock与key的内容，然后检查key的状态；有进程在临界区时，重复交换和检查过程，直到其他进程退出时，检查通过。图 3-17显示的是所有要访问临界资源的进程的相关代码。

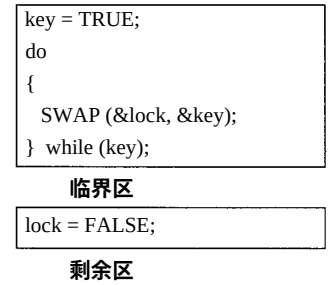


图3-17 互斥算法（Swap指令）

与前面的软件方法相比，硬件方法由于采用处理器指令很好地把修改和检查操作结合成一个不可分的整体而具有明显的优点。具体而言，硬件方法的优点体现在以下几个方面：① 适用范围广：硬件方法适用于任意数目的进程，在单处理器和多处理器环境中完全相同；② 简单：硬件方法的标志设置简单，含义明确，容易验证其正确性；③ 支持多个临界区：在一个进程内有多个临界区时，只需为每个临界区设立一个布尔变量。

硬件方法有许多优点，但也有一些自身无法克服的缺点。这些缺点主要包括：① 进程在等待进入临界区时要耗费处理器时间，不能实现“让权等待”；② 由于进入临界区的进程是从等待进程中随机选择的，有的进程可能一直选不上，从而导致“饥饿”。

### 3.4.2 信号量

前面的互斥算法都是平等进程间的协商机制，它们存在的问题是平等协商无法解决的，需要引入一个地位高于进程的管理者来解决公有资源的使用问题。操作系统可以从进程管理者的角度来处理互斥的问题，信号量(semaphore)就是由操作系统提供的管理公有资源的有效手段。信号量代表可用资源实体的数量。

1. 信号量和P、V原语

信号量是荷兰学者Dijkstra于1965年提出的一种卓有成效的进程同步机制。信号量机制所使用的P、V原语就来自荷兰语的test(proberen)和increment(verhogen)。每个信号量s除一个整数值s.count（计数）外，还有一个进程等待队列s.queue，其中是阻塞在该信号量的各个进程的标识。信号量只能通过初始化和两个标准的原语来访问。作为操作系统核心代码的一部分，P、V原语的执行，不受进程调度和执行的打断，从而很好地解决了原语操作的整体性的问题。信号量的初始化可指定一个非负整数值，表示空闲资源总数。若信号量为非负整数值，该值表示当前的空闲资源数；若为负值，其绝对值表示当前等待临界区的进程数。

依据我们对临界区访问过程的分析，信号量机制中P原语相当于进入区操作，V原语相当于退出区操作。下面我们分析操作系统对这两个原语操作的处理过程。

P原语所执行的操作可用下面函数wait(s)来描述。

```
wait(s)
{
--s.count;                  //表示申请一个资源；
if (s.count <0)//表示没有空闲资源；
    {
    调用进程进入等待队列s.queue;
    阻塞调用进程；
    }
    }
```

V原语所执行的操作可用下面函数signal(s)来描述。

```
signal(s)
{
++s.count;                  //表示释放一个资源；
if (s.count <= 0)    //表示有进程处于阻塞状态；
    {
    从等待队列s.queue中取出头一个进程P;
    进程P进入就绪队列；
    }
}
```

利用操作系统提供的信号量机制，可实现临界资源的互斥访问。如图3-18所示，我们为临界资源设置一个互斥信号量mutex(MUTual Exclusion)，其初值为1；在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间。

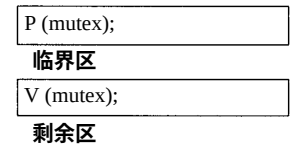


图3-18　互斥算法（信号量）

在使用信号量进行共享资源访问控制时，必须成对使用P、V原语。遗漏P原语则不能保证互斥访问，遗漏V原语则不能在使用临界资源之后将其释放给其他等待的进程。P、V原语的使用不能次序错误、重复或遗漏。

利用操作系统提供的信号量机制，可实现进程间的同步，即所谓的前趋关系。如图3-19所示，

前趋关系是指并发执行的进程$P_1$和$P_2$中，分别有代码$C_1$和$C_2$，要求$C_1$在$C_2$开始前完成执行。我们可为每个前趋关系设置一个互斥信号量$S_{12}$，其初值为0。这样，只有在$P_1$执行到$V(S_{12})$后，$P_2$才会结束$P(S_{12})$的执行。

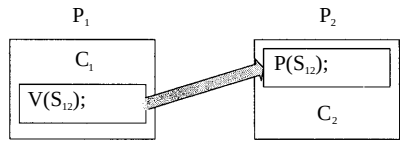


图3-19　利用信号量描述前趋关系

2. 信号量集

当利用信号量机制解决了单个资源的互斥访问后，我们讨论如何控制同时需要多个资源的互斥访问。信号量集是指同时需要多个资源时的信号量操作。

(1) AND型信号量集

AND型信号量集是指同时需要多个资源且每种占用一个资源时的信号量操作。当一段处理代码需要同时获取两个或多个临界资源时，就可能出现由于各进程分别获得部分临界资源并等待其余的临界资源的局面。各进程都会“各不相让”，从而出现死锁。解决这个问题的一个基本思路是：在一个原语中申请整段代码需要的多个临界资源，要么全部分配给它，要么一个都不分配给它。这就是AND型信号量集的基本思想。我们称AND型信号量集P原语为Swait (Simultaneous Wait)，V原语为Ssignal (Simultaneous Signal)。在Swait时，各个信号量的次序并不重要，虽然会影响进程归入哪个阻塞队列，但是因为是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。下面是Swait和Ssignal的伪代码。

```
Swait(S1, S2, ..., Sn)  //P原语；
{
while (TRUE)
  {
  if (S1 >=1 && S2 >= 1 && ... && Sn >= 1)
    {          //满足资源要求时的处理；
    for (i = 1; i <= n; ++i) Si--;
        //注：与wait的处理不同，这里是在确信可满足资源要求时，才进行减 1操作；
        break;
    }
else
    {          //某些资源不够时的处理；
    调用进程进入第一个小于1信号量的等待队列Sj.queue;
    阻塞调用进程；
    }
  }
}

Ssignal(S1, S2, ..., Sn)//V原语；
{
for (i = 1; i <= n; ++i)
  {
    ++Si;              //释放占用的资源；
    for (each process P waiting in Si.queue)
```

```
        //检查每种资源的等待队列的所有进程；
      {
      从等待队列Si.queue中取出进程P;
      if （判断进程P是否通过Swait中的测试）
         //注：与signal不同，这里要进行重新判断；
        { //通过检查（资源够用）时的处理；
        进程P进入就绪队列；
        }
      else
        { //未通过检查（资源不够用）时的处理；
        进程P进入某等待队列；
        }
      }
    }
  }
```

(2) 一般“信号量集”

一般“信号量集”是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。由于一次需要 n个某类临界资源，因此如果通过 n次wait操作申请这n个临界资源，操作效率很低，并可能出现死锁。一般信号量集的基本思路就是在 AND型信号量集的基础上进行扩充，在一次原语操作中完成所有的资源申请。进程对信号量 $S_i$的测试值为 $t_i$（表示信号量的判断条件，要求 $S_i >= t_i$；即当资源数量低于$t_i$时，便不予分配），占用值为$d_i$（表示资源的申请量，即$S_i = S_i - d_i$）。对应的P、V原语格式为：

```
Swait(S1, t1, d1; ...; Sn, tn, dn);
Ssignal(S1, d1; ...; Sn, dn);
```

一般“信号量集”可以用于各种情况的资源分配和释放。下面是几种特殊的情况：

1) Swait(S, d, d)表示每次申请d个资源，当资源数量少于d个时，便不予分配。

2) Swait(S, 1, 1)表示互斥信号量。

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

由于一般信号量在使用时的灵活性，因此通常并不成对使用 Swait和Ssignal。为了避免死锁，可一起申请所有需要的资源，但不一起释放。

### 3.4.3　经典进程同步问题

本节我们讨论几个利用信号量来实现进程互斥和同步的经典例子。这里的主要问题是如何选择信号量和如何安排P、V原语的使用顺序。

依据信号量与进程的关系，我们可把进程中使用的信号量分成私有信号量和公用信号量。私有信号量是指只与制约进程和被制约进程有关的信号量；公用信号量是指与一组并发进程有关的信号量。

1. 生产者—消费者问题

**生产者—消费者问题**(producer-consumer problem)**是指若干进程通过有限的共享缓冲区交换数据时的缓冲区资源使用问题。假设“生产者”进程不断向共享缓冲区写入数据（即生产数据），而“消费者”进程不断从共享缓冲区读出数据（即消费数据）；共享缓冲区共有** n**个；任何时刻只能有一个进程可对共享缓冲区进行操作。所有生产者和消费者之间要协调，以完成对共享缓冲区的操作。如图**3-20**所示。**

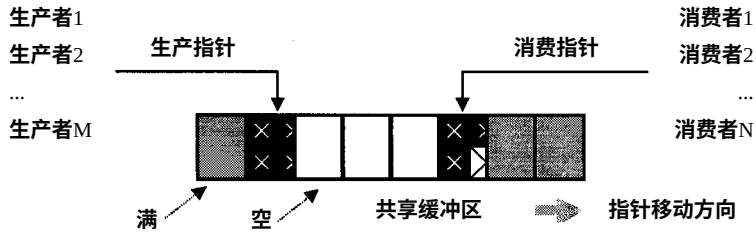


**图**3-20　**生产者—消费者问题**

**我们可把共享缓冲区中的** n**个缓冲块视为共享资源，生产者写入数据的缓冲块成为消费者的可用资源，而消费者读出数据后的缓冲块成为生产者的可用资源。为此，可设置三个信号量：**full、empty**和**mutex**。其中：**full**表示有数据的缓冲块数目，初值是** 0**；**empty**表示空的缓冲块数目，初值是**n**；**mutex**用于访问缓冲区时的互斥，初值是** 1**。实际上，**full**和**empty**间存在如下关系：**full + empty == n**。图**3-21**所示为生产者与消费者的算法。**

生产者

```
P(empty);
P(mutex);    //进入区
  one unit --> buffer;
V(mutex);
V(full);     //退出区
```

消费者

```
P(full);
P(mutex);    //进入区
  one unit <-- buffer;
V(mutex);
V(empty);    //退出区
```

**图**3-21　**利用信号量的生产者与消费者算法**

**注意：这里每个进程中各个** P**操作的次序是重要的。各进程必须先检查自己对应的资源数目，在确信有可用资源后再申请对整个缓冲区的互斥操作；否则，先申请对整个缓冲区的互斥操作，后申请自己对应的缓冲块资源，就可能死锁。出现死锁的条件是，申请到对整个缓冲区的互斥操作后，才发现自己对应的缓冲块资源，这时已不可能放弃对整个缓冲区的占用。**

**如果采用** AND**信号量集，相应的进入区和退出区都很简单。如生产者的进入区为** Swait(empty, mutex)**，退出区为** Ssignal(full, mutex)**。**

2. 读者—写者问题

**读者—写者问题**(readers-writers problem)**是指多个进程对一个共享资源进行读写操作的问题。假设“读者”进程可对共享资源进行读操作，“写者”进程可对共享资源进行写操作；任一时刻“写者”最多只允许一个，而“读者”则允许多个。即对共享资源的读写操作限制关系包括：“读**

—写”互斥、“写—写”互斥和“读—读”允许。

我们可认为写者之间、写者与第一个读者之间要对共享资源进行互斥访问，而后续读者不需要互斥访问。为此，可设置两个信号量 Wmutex、Rmutex和一个公共变量 Rcount。其中：Wmutex表示“允许写”，初值是 1；公共变量 Rcount表示“正在读”的进程数，初值是 0；Rmutex表示对Rcount的互斥操作，初值是1。图3-22所示为读者—写者算法。

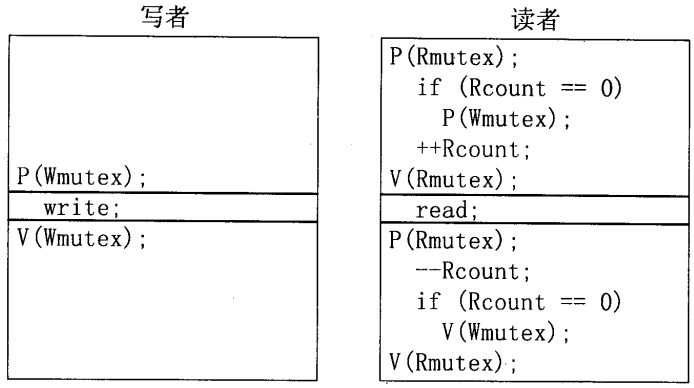


图3-22　读者—写者算法（信号量）

在这个例子中，我们可见到临界资源访问过程的嵌套使用。在读者算法中，进入区和退出区又分别嵌套了一个临界资源访问过程。

对读者—写者问题，也可采用一般“信号量集”机制来实现。如果我们在前面的读写操作限制上再加一个限制条件：同时读的“读者”最多R个。这时，可设置两个信号量Wmutex和Rcount。其中：Wmutex表示“允许写”，初值是1；Rcount表示“允许读者数目”，初值为R。图3-23所示为采用一般“信号量集”机制来实现的读者—写者算法。

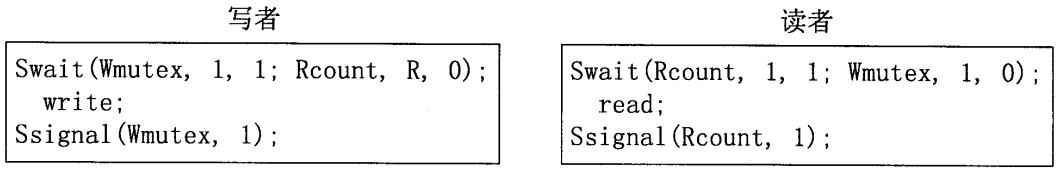


图3-23　读者—写者算法（信号量集）

### 3.4.4　管程

用信号量可实现进程间的同步和互斥，但由于信号量的控制分布在整个程序中，其正确性分析很困难。管程就是为了解决信号量机制面临的困难而提出的一种新的进程间同步机制，它把对信号量的控制集中在管程内部，保证进程互斥地访问共享变量，并方便地阻塞和唤醒进程。相比之下，管程比信号量好控制。

首先，我们来分析一下信号量机制存在的问题。从前面的例子可以观察到，信号量机制的同

步操作是分散在各进程中的，使用不当就可能导致各进程死锁（如 P、V操作的次序错误、重复或遗漏）。使用信号量机制的程序代码易读性差，要了解对一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发执行的多个程序。信号量机制的使用不利于代码的修改和维护，各模块的独立性差，任一组变量或一段代码的修改都可能影响全局。信号量机制的正确性难以保证，操作系统或并发程序通常会使用很多信号量，它们的关系错综复杂，很难保证这样一个复杂的系统没有逻辑错误。

为了解决信号量机制的这些问题， Hoare和Hanson于1973年提出了管程机制。管程的基本思想是把信号量及其操作原语封装在一个对象内部。即，将共享资源以及针对共享资源能够进行的所有操作集中在一个模块中。可把“管程”(monitor)定义为关于共享资源的数据结构以及一组针对该资源的操作过程所构成的软件模块。管程可以以函数库的形式实现，一个管程就是一个基本程序单位，可以单独编译。管程中引入了面向对象的思想，一个管程不仅有关于共享资源的数据结构，而且还有对数据结构进行操作的代码。管程对共享资源进行了封装，进程可调用管程中定义的操作过程，而这些操作过程的实现，在管程外部是不可见的。

引入管程可得到多方面的好处。首先，由于管程的良好封装特征，因此可增强模块的独立性。按资源管理的观点，可把系统分解成若干模块，用数据表示抽象系统资源；同时利用共享资源和专用资源在管理上的差别，按不同的管理方式定义模块的类型和结构，使同步操作相对集中，从而增加了模块的相对独立性。其次，引入管程可提高代码的可读性，便于修改和维护，正确性易于保证。由于采用集中式同步机制，一个操作系统或并发程序由若干个这样的模块所构成，一个模块通常较短，模块之间关系清晰，有利于正确性的保证。

在管程的实现中，为了保证管程共享变量的数据完整性，需要保证管程的互斥进入，并在管程中设置进程等待队列以及相应的等待及唤醒操作，进程只能通过调用管程中所说明的外部过程来间接地访问管程中的共享变量。当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权；当一个进入管程的进程执行唤醒操作时（如P唤醒Q），管程中就存在两个同时处于活动状态的进程。为此，每个管程都设置有一个入口等待队列和一个紧急等待队列。当一个进程试图进入一个已被占用的管程时，它在管程的入口处等待，这个在管程的入口处的进程等待队列称作入口等待队列。在管程内部，由于执行唤醒操作，因此可能会出现多个等待进程，它们已被唤醒，但由于管程的互斥进入而等待，这个等待队列称为紧急等待队列。紧急等待队列的优先级高于入口等待队列的优先级。

在此，我们需要说明管程与进程的区别。在操作系统中设置进程是为了描述程序的动态执行过程，而设置管程是为了进行进程的同步，协调进程的相互关系和对共享资源进行访问。操作系统维护的进程数据结构是进程控制块，而与管程相关的数据结构是等待队列。管程可被进程调用。管程与操作系统中的共享资源相关，没有创建和撤消；而进程有创建和撤消。

### 3.4.5 Windows 2000/XP的进程互斥和同步

在Windows 2000/XP中提供了互斥对象、信号量对象和事件对象三种同步对象和相应的系统调用，用于进程和线程的同步。这些同步对象都有一个用户指定的对象名称，不同进程中用同样

的对象名称来创建或打开对象，从而获得该对象在本进程的句柄。从本质上讲，这组同步对象的功能是相同的，它们的区别在于适用场合和效率会有所不同。

互斥对象(Mutex)就是互斥信号量，在一个时刻只能被一个线程使用。它的相关 API包括：CreateMutex、OpenMutex和ReleaseMutex。CreateMutex创建一个互斥对象，返回对象句柄；OpenMutex打开并返回一个已存在的互斥对象句柄，用于后续访问；而 ReleaseMutex释放对互斥对象的占用，使之成为可用。

信号量对象(Semaphore)就是资源信号量，初始值的取值在 0到指定最大值之间，用于限制并发访问的线程数。它的相关 API包括：CreateSemaphore、OpenSemaphore和ReleaseSemaphore。CreateSemaphore创建一个信号量对象，在输入参数中指定最大值和初值，返回对象句柄；OpenSemaphore返回一个已存在的信号量对象句柄，用于后续访问； ReleaseSemaphore释放对信号量对象的占用。

事件对象(Event)相当于“触发器”，可用于通知一个或多个线程某事件的出现。它的相关 API包括：CreateEvent、OpenEvent、SetEvent、ResetEvent和PulseEvent。CreateEvent创建一个事件对象，返回对象句柄； OpenEvent返回一个已存在的事件对象句柄，用于后续访问； SetEvent和PulseEvent设置指定事件对象为可用状态；ResetEvent设置指定事件对象为不可用状态。

对于这三种同步对象， Windows 2000/XP提供了两个统一的等待操作 WaitForSingleObject和WaitForMultipleObjects。WaitForSingleObject可在指定的时间内等待指定对象为可用状态；WaitForMultipleObjects可在指定的时间内等待多个对象为可用状态。这两个 API的接口为：

```
DWORD WaitForSingleObject( HANDLE hHandle等待对象句柄；
   DWORD dwMilliseconds     // 以毫秒为单位的最长等待时间；
   );
DWORD WaitForMultipleObjects( DWORD nCount,
   //对象句柄数组中的句柄数；
   CONST HANDLE *lpHandles,
   // 指向对象句柄数组的指针，数组中可包括多种对象句柄；
   BOOL bWaitAll,
   // 等待标志：TRUE表示所有对象同时可用，FALSE表示至少一个对象可用；
   DWORD dwMilliseconds 等待超时时限；
   );
```

除了上述三种同步对象， Windows 2000/XP还提供了一些与进程同步相关的机制，如临界区对象和互锁变量访问 API等。临界区(Critical Section)对象只能用于在同一进程内使用的临界区，同一进程内各线程对它的访问是互斥进行的。把变量说明为 CRITICAL_SECTION类型，就可作为临界区使用。相关 API包括：InitializeCriticalSection、EnterCriticalSection、TryEnterCriticalSection、LeaveCriticalSection和DeleteCriticalSection。InitializeCriticalSection对临界区对象进行初始化；EnterCriticalSection等待占用临界区的使用权，得到使用权时返回； TryEnterCriticalSection非等待方式申请临界区的使用权，申请失败时返回 0；LeaveCriticalSection释放临界区的使用权；DeleteCriticalSection释放与临界区对象相关的所有系统资源。

互锁变量访问API相当于硬件指令，用于对整型变量的操作，可避免线程间切换对操作连续性的影响。这组互锁变量访问 API包括：InterlockedExchange、InterlockedCompareExchange、InterlockedExchangeAdd、InterlockedDecrement、InterlockedIncrement。InterlockedExchange进行32位数据的先读后写原子操作；InterlockedCompareExchange依据比较结果进行赋值的原子操作；InterlockedExchangeAdd先加后存结果的原子操作；InterlockedDecrement先减1后存结果的原子操作；InterlockedIncrement先加1后存结果的原子操作。

## 3.5 进程间通信

进程间通信(Inter-Process Communication, IPC)要解决的问题是进程间的信息交流。这种信息交流的量可大可小。操作系统提供了多种进程间通信机制，可分别适用于多种不同的场合。前面所介绍的进程同步机制实际上就是进程间通信的一种，只不过交流的信息量非常少。

按通信量的大小，我们可把进程间通信分成低级通信和高级通信。在低级通信中，进程间只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量机制。它的优点是速度快，缺点是传送信息量小、通信效率低、编程复杂。由于每次通信传递的信息量固定，因此如果传递较多信息则需要进行多次通信。用户直接实现通信的细节，编程复杂，容易出错。在高级通信中，进程间可传送任意数量的数据，包括共享存储区、管道和消息等机制。

按通信过程中是否有第三方作为中转，我们可把进程间通信分成直接通信和间接通信。直接通信是指发送方把信息直接传递给接收方。在直接通信方式中，发送方要指定接收方的地址或标识，也可以指定多个接收方或广播式地址；接收方可接收来自任意发送方的消息，并在读出消息的同时获取发送方的地址。间接通信是指通信过程要借助于收发双方进程之外的共享数据结构（如消息队列）作为通信中转。在间接通信方式中，接收方和发送方的数目可以是任意的。

进程间通信还要考虑到通信过程中的一些其他特征，如通信链路特征、数据格式和收发双方的同步方式等。对进程间通信模式有影响的通信链路特征包括：链路是点对点还是广播链路；通信链路是否带缓冲区；链路是单向还是双向等。数据格式主要分成字节流和报文两类。采用字节流格式时，接收方不保留各次发送之间的分界；采用报文格式时，接收方保留各次发送之间的分界。报文方式还可进一步分成定长报文 /不定长报文和可靠报文 /不可靠报文。收发操作的同步方式可分成阻塞和不阻塞两种。阻塞操作是指操作方要等待操作结束；不阻塞操作是指操作提交后立即返回。

### 3.5.1 Windows 2000/XP的信号

信号(signal)是进程与外界的一种低级通信方式，相当于进程的“软件”中断。进程可发送信号，每个进程都有指定信号处理例程。信号通信是单向的和异步的。 Windows 2000/XP有两组与信号相关的系统调用，分别处理不同的信号。

1. SetConsoleCtrlHandler和GenerateConsoleCtrlEvent

SetConsoleCtrlHandler可定义或取消本进程的信号处理例程(HandlerRoutine)列表中的用户定义例程。例如，缺省时，每个进程都有一个信号 CTRL+C的处理例程，我们可利用 SetConsoleCtrl

Handler调用来忽视或恢复对CTRL+C的处理。GenerateConsoleCtrlEvent可发送信号到与本进程共享同一控制台的控制台进程组。这一组系统调用处理的信号包括表3-3中的5种信号。

表　3-3

| 信 号 名 | 说　明 |
|---|---|
| CTRL_C_EVENT | 收到CTRL+C信号 |
| CTRL_BREAK_EVENT | 收到CTRL+BREAK信号 |
| CTRL_CLOSE_EVENT | 当用户关闭控制台时系统向该控制台的所有进程发送的控制台关闭信号 |
| CTRL_LOGOFF_EVENT | 用户退出系统时系统向所有控制台进程发送的退出信号 |
| CTRL_SHUTDOWN_EVENT | 系统关闭时系统向所有控制台进程发送的关机信号 |

2. signal和raise

signal用于设置中断信号处理例程。raise用于发送信号。这一组系统调用处理的信号包括表3-4中列出的6种信号。这6种信号是与传统的UNIX系统相同的，而前面一组系统调用处理的5种信号是Windows 2000/XP中特有的。

表　3-4

| 信　号　名 | 说　明 |
|---|---|
| SIGABRT | 非正常终止 |
| SIGFPE | 浮点计算错误 |
| SIGILL | 非法指令 |
| SIGINT | CTRL+C信号(对Win32无效) |
| SIGSEGV | 非法存储访问 |
| SIGTERM | 终止请求 |

### 3.5.2　Windows 2000/XP基于文件映射的共享存储区

共享存储区(shared memory)可用于进程间的大数据量通信。进行通信的各进程可以任意读写共享存储区，也可在共享存储区上使用任意数据结构。在使用共享存储区时，需要进程互斥和同步机制的辅助来确保数据一致性。Windows 2000/XP采用文件映射(file mapping)机制来实现共享存储区，用户进程可以将整个文件映射为进程虚拟地址空间的一部分来加以访问。

下面系统调用与共享存储区的使用相关。CreateFileMapping为指定文件创建一个文件映射对象，返回对象指针；OpenFileMapping打开一个命名的文件映射对象，返回对象指针；MapViewOfFile把文件映射到本进程的地址空间，返回映射地址空间的首地址；FlushViewOfFile可把映射地址空间的内容写到物理文件中；UnmapViewOfFile拆除文件与本进程地址空间之间的映射关系；CloseHandle可关闭文件映射对象。当完成文件到进程地址空间的映射后，就可利用首地址进行读写。在信号量等机制的辅助下，通过一个进程向共享存储区写入数据而另一个进程从共享存储区读出数据，就可在两个进程间实现大量数据的交流。

### 3.5.3 Windows 2000/XP管道

管道(pipe)是一条在进程间以字节流方式传送的通信通道。它是利用操作系统核心的缓冲区（通常几十个KB）来实现的一种单向通信，常用于命令行所指定的输入输出重定向和管道命令。在使用管道前要建立相应的管道，然后才可使用。

Windows 2000/XP提供无名管道和命名管道两种管道机制。 Windows 2000/XP的无名管道类似于UNIX系统的管道，但提供的安全机制比 UNIX管道完善。利用 CreatePipe可创建无名管道，并得到两个读写句柄；然后利用 ReadFile和WriteFile可进行无名管道的读写。下面是 CreatePipe的调用格式。

```
BOOL CreatePipe( PHANDLE hReadPipe,读句柄；
PHANDLE hWritePipe, 写句柄；
LPSECURITY_ATTRIBUTES lpPipeAttributes安全属性指针；
DWORD nSize   // 管道缓冲区字节数；
);
```

Windows 2000/XP的命名管道是服务器进程与一个客户进程间的一条通信通道，可实现不同机器上的进程通信。它采用客户-服务器模式连接本机或网络中的两个进程。在建立命名管道时，存在一定的限制。即服务器方 (创建命名管道的一方 )只能在本机上创建命名管道，命名方式只能是“ \\.\pipe\PipeName ”形式，不能在其他机器上创建管道；但客户方（连接到一个命名管道实例的一方）可以连接到其他机器上的命名管道，命名方式可为“ \\serverName\pipe\pipename ”形式。服务器进程为每个管道实例建立单独的线程或进程。下面是与命名管道相关的主要系统调用。CreateNamedPipe在服务器端创建并返回一个命名管道句柄； ConnectNamedPipe在服务器端等待客户进程的请求；CallNamedPipe从管道客户进程建立与服务器的管道连接； ReadFile、WriteFile（用于阻塞方式）、ReadFileEx、WriteFileEx（用于非阻塞方式）用于命名管道的读写。

### 3.5.4 Windows 2000/XP邮件槽

Windows 2000/XP中提供的邮件槽(mailslot)是一种不定长、不可靠的单向消息通信机制。消息的发送不需要接收方准备好，随时可发送。邮件槽也采用客户-服务器模式，只能从客户进程发往服务器进程。服务器进程负责创建邮件槽，它可从邮件槽中读消息；而客户进程可利用邮件槽的名字向它发送消息。在建立邮件槽时，也存在一定的限制。即服务器进程（接收方）只能在本机建立邮件槽，命名方式只能是“ \\.\mailslot\[path]name ”方式；但客户进程（发送方）可打开其他机器上的邮件槽，命名方式可为“ \\range\mailslot\[path]name ”，这里range可以是本机、其他机器的名字或域名。下面是与邮件槽相关的主要系统调用。 CreateMailslot服务器方创建邮件槽，返回其句柄；GetMailslotInfo服务器查询邮件槽的信息，如消息长度、消息数目、读操作等待时限等；SetMailslotInfo服务器设置读操作等待时限； ReadFile服务器读邮件槽；CreateFile客户方打开邮件槽；WriteFile客户方发送消息。由于邮件槽不提供可靠的传输机制，因此在邮件槽

关闭过程中可能出现信息丢失的情况。在邮件槽的所有服务器句柄关闭后，邮件槽被关闭，如果这时还有未读出的消息，这些消息将会被丢弃，所有客户句柄都被关闭。

### 3.5.5　套接字

套接字(socket)是一种网络通信机制，它通过网络在不同计算机上的进程间进行双向通信。套接字所采用的数据格式可为可靠的字节流或不可靠的报文，通信模式可为客户　/服务器模式或对等模式。为了实现不同操作系统上的进程通信，需要约定网络通信时不同层次的通信过程和信息格式，TCP/IP协议就是广泛使用的网络通信协议。

在UNIX系统中使用的BSD套接字主要是基于TCP/IP协议，操作系统中有一组标准的系统调用完成通信连接的维护和数据收发。例如，send和sendto用于数据发送，而recv和recvfrom用于数据接收。

Windows 2000/XP中的套接字规范称为“ Winsock"，它除了支持标准的BSD套接字外，还实现了一个真正与协议独立的应用程序编程接口，可支持多种网络通信协议。例如，在　WinSock 2.2中分别把 send、sendto、recv和recvfrom扩展成WSASend、WSASendto、WSARecv和WSARecvfrom。

## 3.6　死锁问题

死锁(deadlock)是指系统中多个进程无限制地等待永远不会发生的条件。在这一节中，我们将讨论如何在操作系统中处理死锁问题。

### 3.6.1　概述

产生死锁的根本原因是对互斥资源的共享，并发执行进程的同步关系不当。为了仔细分析死锁的形成过程，我们把进程使用的资源分成可重用资源和可消耗资源两类。对于可重用资源(reusable resource)，每个时刻只有一个进程使用，在宏观上各个进程轮流使用。如处理器、主存和辅存、I/O通道、外设、数据结构（如文件、数据库和信号量等）都是可重用资源。在可重用资源使用时出现的死锁，都是由于各进程都拥有部分资源，同时在请求其他进程已占有的资源，从而造成永远等待。例如，假设系统中的资源　A和B都只有一个，则如图 3-24所示的申请次序“ P1<a> P2<a> P1<b> P2<b> ”就会形成死锁。

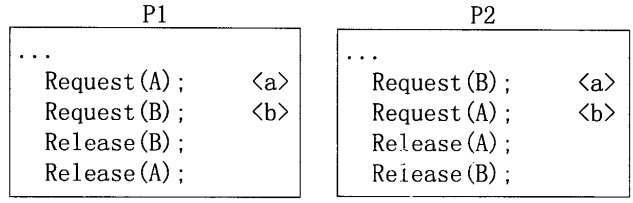


图3-24　可重用资源死锁

可消耗资源(consumable resource)是指可以动态生成和消耗资源，一般不限制数量。如硬件

中断、信号、消息、缓冲区内的数据等都是可消耗资源。由于可消耗资源的生成和消耗存在依赖关系，因此它们的使用也可能因为双方都等待对方生成资源而形成死锁。如图 3-25所示的执行次序“ P1<a> P2<a> ”就会由于对方还未生成资源而形成永远等待。

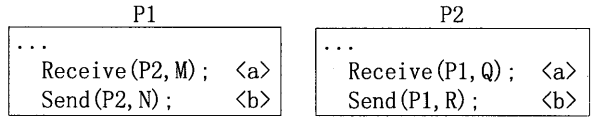


图3-25 可消耗资源死锁

由上面的分析可知，当多个进程并发执行时，由于相互的推进顺序的不确定，很可能会导致死锁。为此，我们需要讨论死锁的形成原因和可能的解决方案。

我们可把死锁发生条件描述成下列四个条件。它们是产生死锁的充分必要条件，只有四个条件都满足时，才会出现死锁。

1) 互斥：任一时刻只允许一个进程使用资源。

2) 请求和保持：进程在请求其余资源时，不主动释放已经占用的资源。

3) 非剥夺：进程已经占用的资源，不会被强制剥夺。

4) 环路等待：环路中的每一条边是进程在请求另一进程已经占有的资源。

在分析了产生死锁的四个条件后，我们可使用表 3-5中的三种基本方法来处理死锁。下面我们将具体讨论如何使用这三种方法来处理死锁问题。

表 3-5

| 方 法 | 资源分配策略 | 各种可能模式 | 主要优点 | 主要缺点 |
|---|---|---|---|---|
| 预防(Prevention) | 保守的；宁可资源闲置（从机制上使死锁条件不成立） | 一次请求所有资源<条件2> | 适用于作为突发式处理的进程；不必剥夺 | 效率低；进程初始化时间延长 |
| | | 资源剥夺<条件3> | 适用于状态可以保存和恢复的资源 | 剥夺次数过多；多次对资源重新起动 |
| | | 资源按序申请<条件4> | 可以在编译时（而不必在运行时）就进行检查 | 不便灵活申请新资源 |
| 避免(Avoidance) | 是“预防”和“检测”的折衷（在运行时判断是否可能死锁） | 寻找可能的安全的运行顺序 | 不必进行剥夺 | 使用条件：必须知道将来的资源需求；进程可能会长时间阻塞 |
| 检测(Detection) | 宽松的；只要允许，就分配资源 | 定期检查死锁是否已经发生 | 不延长进程初始化时间；允许对死锁进行现场处理 | 通过剥夺解除死锁，造成损失 |

### 3.6.2 死锁的预防

预防死锁是指通过某种策略来限制并发进程对资源的请求，使系统在任何时刻都不满足死锁

的必要条件。预先静态分配法和有序资源使用法是两种预防死锁的基本策略。针对死锁的第 2个条件，预先静态分配法通过预先分配进程运行所需的全部资源，从而保证进程在运行过程中不等待资源。这种做法降低了对资源的利用率，降低了进程的并发程度；并且只适用于有可能预先知道所需资源的情况下，但在实际的进程运行过程中有可能无法预先知道所需的资源。针对死锁的第4个条件，有序资源使用法把资源分类按顺序排列，从而保证资源的申请不形成环路。这种做法会限制进程对资源的请求顺序，同时资源的排序占用系统开销。

### 3.6.3　死锁的检测

检测死锁的基本思路是在操作系统中保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。死锁检测算法主要是检查是否有循环等待。我们可把进程和资源间的申请和分配关系描述成一个有向图，通过检查有向图中是否有循环来判断死锁的存在，这就是死锁检测的资源分配图(resource allocation graph)算法。

有向图G的顶点为资源或进程，我们定义从资源 R到进程P的边表示R已分配给P，从进程P到资源R的边表示P正因请求R而处于等待状态。如果有向图中存在循环，则表示死锁的存在。为了在复杂的有向图中判断是否有循环，我们可通过资源分配图的等价变换来简化有向图。具体简化过程如下：① 删除不处于等待状态的进程（即没有从该进程出发的边）；② 依次删除当前的叶顶点。可以证明，简化后还存在边的不可简化的资源分配图存在死锁，其中的有边进程为死锁进程。

资源分配图算法可检测死锁，但还需要进一步的操作来解除死锁。解除死锁时，常常造成进程终止或重新起动。我们需要选择或挂起或终止哪一个进程。通常是选择撤消或挂起代价最小的死锁进程，并抢占它的资源，以解除死锁。所谓代价最小的判断原则可为进程优先级或系统会计过程给出的运行代价。

### 3.6.4　死锁的避免

解决死锁问题的最合理做法应是在分配资源时判断是否会出现死锁，只在确信不会导致死锁时才分配资源。这就是死锁的避免。避免死锁的主要困难在于，我们在很多时候无法判断进程的资源申请是否会导致死锁或判断代价过高。下面我们介绍一种死锁避免算法，即银行家算法。

所谓银行家问题是指银行家在向顾客贷款时如何保证资金的安全。我们假设一个银行家把他的固定资金贷给若干顾客时，只要不出现一个顾客借走所有资金后还不够，银行家的资金就是安全的。银行家需要一个算法保证借出去的资金在有限时间内可收回。

假定顾客分成若干次进行贷款，并在第一次借款时能说明他的最大借款额。在这种假定条件下，银行家算法就可保证资金的安全。具体操作过程如下：顾客的借款操作是依次顺序进行的，直到全部操作完成。银行家对当前顾客的借款操作进行判断：银行能否支持顾客借款，直到全部归还。如果能，则本次贷款是安全的；否则就是不安全的。当判断结果为安全时执行贷款；否则暂不贷款。

由于银行家算法允许资源的部分分配和不需要抢占，因此在操作系统中采用该算法可提高资源利用率。但采用银行家算法的前提太严格，要求事先说明最大资源要求量，这在现实中是很难实现的。

### 3.6.5 解决死锁问题的综合方法

基于以上分析，我们在实际操作系统中不可能单纯采用某一种方法来解决死锁问题，现实的做法是多种方法的综合使用。例如，首先，我们可对资源进行分类，将各种资源归入若干个不同的资源类中，如外存交换区空间资源、外部设备资源、内存资源等。其次，对资源类进行排序，在不同资源类之间规定次序，对不同资源类中的资源采用线性按序申请的方法。进一步，我们还可对同一资源类中的资源进行针对性处理，采用不同的适当方法。例如，使用避免方法处理外设资源分配，而对存储资源则采用预防方法。

## 3.7 处理器调度概述

在下面几节中，我们将讨论处理器资源的管理问题，即如何在进程或线程间分配和回收处理器执行时间。处理器是计算机系统中的重要资源，处理器调度算法不仅对处理器的利用效率和用户进程的执行有影响，同时还与内存等其他资源的使用密切相关，对整个计算机系统的综合性能指标有重要影响。

### 3.7.1 处理器调度的类型

从处理器调度的对象、时间、功能等不同角度，我们可把处理器调度分成不同类型。处理器调度不仅涉及选择哪一个就绪进程进入运行状态，还涉及何时启动一个进程的执行。按照调度所涉及的层次的不同，我们可把处理器调度分成宏观调度、中级调度和微观调度三个层次。

宏观调度也称为作业调度或高级调度。从用户工作流程的角度，一次作业提交若干个流程，其中每个程序按照流程进行调度执行。宏观调度的时间尺度通常是分钟、小时或天。中级调度涉及进程在内外存间的交换。从存储器资源管理的角度来看，把进程的部分或全部换出到外存上，可为当前运行进程的执行提供所需的内存空间，将当前进程所需部分换入到内存。指令和数据必须在内存里才能被处理器直接访问。微观调度也称为低级调度。从处理器资源分配的角度来看，处理器需要经常选择就绪进程或线程进入运行状态。微观调度的时间尺度通常是毫秒级的。由于微观调度算法的频繁使用，因此要求在实现时做到高效。在图 3-26中，我们给出了三种层次的处理器调度算法所涉及的进程或线程状态转换。

### 3.7.2 调度的性能准则

我们可从不同的角度来判断处理器调度算法的性能，如用户的角度、处理器的角度和算法实现的角度。实际的处理器调度算法选择是一个综合的判断结果。

处理器调度是为了执行用户程序，必须考虑用户对调度的要求。用户关心的处理器调度性能指标主要包括周转时间、响应时间、公平性和优先级等。周转时间是指作业从提交到完成（得到结果）所经历的时间。周转时间的组成包括进程在收容队列中的等待时间、占用处理器的执行时间、在就绪队列和阻塞队列中的等待进行等。为了去除进程本身因素的影响，在讨论处理器调度时也使用平均周转时间 T和平均带权周转时间作为衡量指标。带权周转时间是指周转时间除以进

程执行时间。响应时间是指用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间。它是分时系统中衡量系统交互性能的指标。在实时系统中，还使用截止时间来衡量系统的实时性能。截止时间可分为开始截止时间和完成截止时间。处理器调度算法的公平性是指调度算法不会因作业或进程本身的特性而使上述指标过分恶化。在处理器调度算法中考虑进程优先级，可以使关键任务得到更好的性能指标。

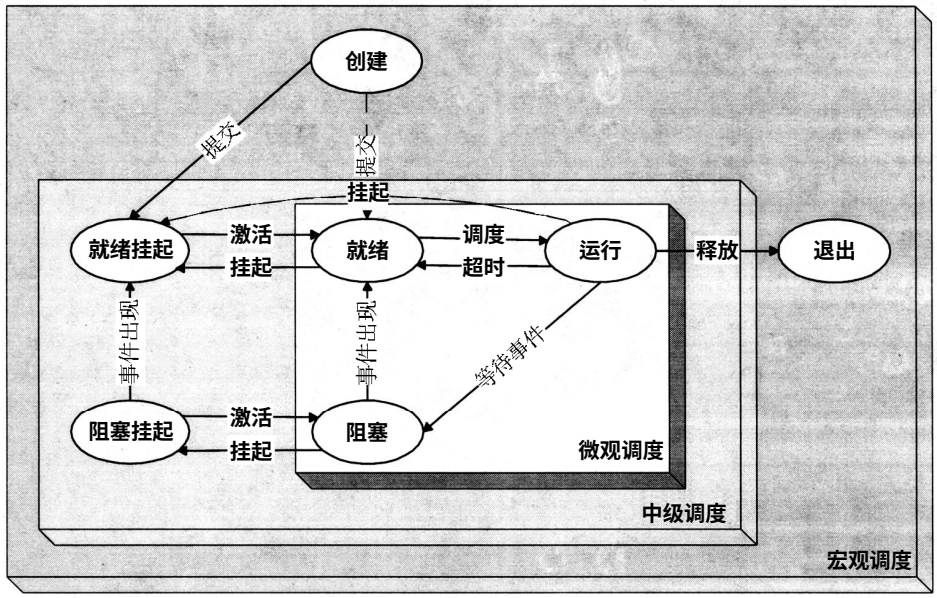


图3-26　处理器调度的层次

操作系统的处理器调度算法选择不仅要考虑用户关心的性能指标，还要考虑系统各部分的利用效率，即面向系统的指标。与系统利用率相关的处理器调度指标包括吞吐量和处理器利用率等。吞吐量是指单位时间内所完成的作业数。这个指标跟作业本身特性和调度算法都有关系。注意，平均周转时间不是吞吐量的倒数，这是因为并发执行的作业在时间上可以重叠。例如，在 2小时内完成4个作业，吞吐量是2个作业/小时；而每个作业的周转时间不一定是半小时。处理器利用率在大中型主机系统中是主要的考虑因素，而在微机系统中重要性相对较低。由于处理器在系统中的主控作用，因此处理器调度算法还会对各种设备的均衡利用效率有影响。例如，在处理器调度算法中考虑到进程的主要操作是计算还是 I/O，并进行有意识的组合，将有利于均衡提高处理器和外设的利用率。

处理器调度算法本身的执行开销和实现困难也是需要考虑的问题。调度算法本身的执行开销不能太复杂，否则无法用于实际操作系统。

### 3.7.3　进程调度器

操作系统为了对进程进行有效的监控，需要维护一些与进程相关的数据结构，记录所有进程

的运行状况，并在进程出让处理器或调度程序剥夺执行状态进程占用的处理器时，选择适当的进程分派处理器，完成上下文切换。我们把操作系统内核中与进程调度相关代码称为进程调度器(dispatcher)。

进程调度器的主要功能是进程状态维护和进程的上下文切换。当一个进程 A进入通过时钟中断或系统调用进入操作系统内核的进程调度器时，首先要保存当前进程 A的上下文，执行调度器代码。在调度器代码的控制下确定是否要进行切换，以及切换到哪个进程。如果要切换到另一进程B，则恢复进程B的上下文，然后开始从上次切换前位置继续执行进程 B代码。

## 3.8 调度算法

在这一节中，我们讨论几种主要的处理器调度算法。由于算法的设计出发点不同，因此它们各自的适用场合也不同。有的算法适用于宏观调度，有的算法适用于微观调度，有的算法则是适用于多种场合。

### 3.8.1 先来先服务算法

先来先服务(First Come First Service, FCFS)算法是最简单的调度算法，它的基本思想是按进程的到达先后顺序进行调度。

FCFS算法按照作业提交或进程变为就绪状态的先后次序来分派处理器；当前作业或进程占用处理器，直到执行完毕或阻塞才让出处理器；当作业或进程唤醒后（如 I/O完成），并不立即恢复执行，通常等到当前作业或进程让出处理器才恢复执行。

FCFS算法的最主要特点是简单。由于它的处理器调度采用非抢占方式，因此操作系统不会强行暂停当前进程的执行， FCFS算法具有下列特点：① 比较有利于长作业，而不利于短作业；② 有利于处理器繁忙的作业，而不利于 I/O繁忙的作业。

### 3.8.2 最短作业优先算法

最短作业优先(Shortest Job First, SJF)又称为最短进程优先(Shortest Process Next, SPN)，它的设计目标是改进FCFS算法，减少进程的平均周转时间。 SJF算法要求作业在开始执行时预计执行时间，对预计执行时间短的作业（进程）优先分派处理器。后来的短作业不抢先正在执行的作业。

由于SJF算法在分派处理时考虑到进程执行时间对周转时间的影响，因而具有如下优点：①与FCFS相比，改善了平均周转时间和平均带权周转时间，缩短了作业的等待时间；② 有利于提高系统的吞吐量。但 SJF算法也存在一些缺点：① 对长作业非常不利，可能长时间得不到执行；② 未能依据作业的紧迫程度来划分执行的优先级；③ 难以准确估计作业（进程）的执行时间，从而影响调度性能。

通过选用其他条件来分派处理器， SJF算法可形成下列变种：① 最短剩余时间优先(Shortest Remaining Time, SRT)，在SJF算法的基础上，该算法允许比当前进程剩余时间更短的进程来抢占；② 最高响应比优先(Highest Response Ratio Next, HRRN)，响应比的定义为“ (等待时间 + 要求执行时间) / 要求执行时间 ”，它是FCFS和SJF的一种折衷。

### 3.8.3　时间片时钟算法

前两种算法主要用于宏观调度，说明怎样选择一个进程或作业开始运行，开始运行后的做法都相同，即运行到结束或阻塞，阻塞结束时等待当前进程放弃处理器 。时间片时钟算法主要用于微观调度，说明怎样并发运行，即切换的方式；它的设计目标是提高资源利用率。其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率。

在时间片时钟(Round Robin)算法中，系统中所有的就绪进程按照FCFS原则，排成一个队列。每次调度时将处理器分派给队首进程，让其执行一个时间片。时间片的长度从几个 ms到几百ms。在一个时间片结束时，发生时钟中断。在时钟中断中，进程调度器暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。进程可以未使用完一个时间片，就让出处理器（如阻塞）。

时间片时钟算法中的时间片长度是影响算法特征的重要因素。我们先考虑两种极端的情况。如果时间片很长，长到大多数进程可在一个时间片内执行完，该算法将退化为 FCFS算法，进程的响应时间长，不能达到提高响应特性的目标。如果时间片过短，用户的一次交互过程也需要多个时间片才能处理完，上下文切换次数增加，响应时间长。因此，时间片长度的选择要与完成一个基本的交互过程所需的时间相当，保证一个基本的交互过程可在一个时间片内完成。我们认为影响时间片长度的主要因素是系统的处理能力和系统的负载状态。依据系统的处理能力确定时间片长度，使用户输入通常在一个时间片内能处理完，否则使响应时间、平均周转时间和平均带权周转时间延长。为了保证不同负载状态下用户交互的响应时间，需要对时间片长度进行适当调整。

### 3.8.4　多级队列算法

多级队列算法(Multiple-level Queue)的基本思想是引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标。在多级队列算法中，根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。每个作业固定归入一个队列，例如，系统进程、用户交互进程、批处理进程等不同队列。不同队列可有不同的优先级、时间片长度、调度策略等。

### 3.8.5　优先级算法

优先级算法(Priority Scheduling)是多级队列算法的改进，协调各进程队列中进程的响应时间要求。优先级算法适用于作业调度和进程调度，可分成抢先式和非抢先式。

优先级算法中各进程的优先级确定方式分为静态和动态两种。静态优先级方式是指在创建进程时确定进程优先级，并保持不变到进程结束。影响进程的静态优先级的主要因素包括进程类型、进程的资源需求和用户要求。通常，系统进程优先级高于其他用户进程，对处理器和内存等资源要求较少的进程优先级较高。进程也可按用户的紧急程度和付费情况等来确定。动态优先级方式是指在创建进程时赋予给进程的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。影响进程动态优先级变化的因素包括进程等待时间和占用处理器时间等。当一个进程在就

绪队列中等待时间越长，它的优先级会越高。这种做法的目的是使优先级较低的进程在等待足够的时间后，其优先级提高，进而被调度执行。当一个进程占用处理器的执行的时间越长，它的优先级就会越低。这种做法的目的是使持续执行的进程会在优先级降低后出让处理器。

### 3.8.6 多级反馈队列算法

多级反馈队列(Round Robin with Multiple Feedback)算法是时间片时钟算法和优先级算法的综合和发展。通过动态调整进程优先级和时间片大小，多级反馈队列算法可兼顾多方面的系统目标。例如，为提高系统吞吐量和缩短平均周转时间而照顾短进程；为获得较好的 I/O设备利用率和缩短响应时间而照顾I/O型进程；同时，也不必事先估计进程的执行时间。

在多级反馈队列算法中，通常设置有多个就绪队列，分别赋予不同的优先级，如队列 1的优先级最高，然后逐级降低。每个队列的执行时间片长度也不同，如规定优先级越低则时间片越长。新进程进入内存后，先投入最高优先级的队列 1的末尾，按FCFS算法调度；如果队列 1的一个时间片未能执行完，则降低投入到队列 2的末尾，同样按FCFS算法调度；如此下去，一直降低到最后的队列，则按时间片时钟算法调度直到完成。仅当较高优先级的队列为空时，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

在实际系统中使用的多级反馈队列算法还可以使用更复杂的动态优先级调整策略。例如，为了保证I/O操作的及时完成，通常会在进程发出 I/O请求后进入最高优先级队列，并执行一个小时间片，以及时响应 I/O交互。对于计算型进程，可在每次执行完一个完整的时间片后，进入更低级队列，并最终采用最大时间片来执行，这样可减少计算型进程的调度次数。对于 I/O次数不多而处理器占用时间较多的进程，可使用高优先级进行 I/O处理，在I/O完成后，放回原来队列，以免每次都从最高优先级队列逐次下降。一种更通用的策略是：进行 I/O时提高优先级；时间片用完时降低优先级。这样可适应一个进程在不同时间段的运行特点。

## 3.9 Windows 2000/XP的线程调度

作为一个实际的操作系统，Windows 2000/XP的处理器调度的调度对象是线程，也称为线程调度。作为一个实际的操作系统，Windows 2000/XP的线程调度并不是单纯使用某一种调度算法，而是多种算法的结合体，根据实际系统的需要进行针对性的优化和改进。下面从 Windows 2000/XP中与处理器调度相关的各个侧面进行深入讨论。首先简要描述 Windows 2000/XP线程调度的主要特征；然后从 Win32应用程序编程接口和 Windows 2000/XP内核的角度讨论优先级；最后说明Windows 2000/XP线程调度的数据结构和调度算法。

### 3.9.1 Windows 2000/XP的线程调度特征

Windows 2000/XP实现了一个基于优先级的抢先式多处理器调度系统。调度系统总是运行优先级最高的就绪线程。通常线程可在任何可用处理器上运行，但可限制某线程只能在某处理器上运行。亲合处理器集合允许用户线程通过 Win32调度函数选择它偏好的处理器。

当一个线程被调度进入运行状态时，它可运行一个被称为时间配额 (quantum)的时间片。时间配额是Windows 2000/XP允许一个线程连续运行的最大时间长度，随后 Windows 2000/XP会中断该线程的运行，判断是否需要降低该线程的优先级，并查找是否有其他高优先级或相同优先级的线程等待运行。 Windows 2000专业版和Windows 2000服务器版的时间配额是不同的，同一系统中各线程的时间配额是可修改的。由于 Windows 2000/XP的抢先式调度特征，因此一个线程的一次调度执行可能并没有用完它的时间配额。如果一个高优先级的线程进入就绪状态，当前运行的线程可能在用完它的时间配额前就被抢先。事实上，一个线程甚至可能在被调度进入运行状态之后开始运行之前就被抢先。

Windows 2000/XP在内核中实现它的线程调度代码，这些代码分布在内核中与调度相关事件出现的位置，并不存在一个单独的线程调度模块。内核中完成线程调度功能的这些函数统称为内核调度器(kernel's dispatcher)。线程调度出现在DPC/线程调度中断优先级。线程调度的触发事件有以下四种：

1) 一个线程进入就绪状态，如一个刚创建的新线程或一个刚刚结束等待状态的线程。

2) 一个线程由于时间配额用完而从运行状态转入退出状态或等待状态。

3) 一个线程由于调用系统服务而改变优先级或被 Windows 2000/XP系统本身改变其优先级。

4) 一个正在运行的线程改变了它的亲合处理器集合。

在这些触发事件出现时， Windows 2000/XP必须选择下一个要运行的线程。当 Windows 2000/XP选择一个新线程进入运行状态时，将执行一个线程上下文切换以使新线程进入运行状态。线程上下文是指保存正在运行线程的相关运行环境，加载另一个线程的相关运行环境，并开始新线程执行的过程。

我们已说过，Windows 2000/XP的处理器调度对象是线程，这时的进程仅作为提供资源对象和线程的运行环境，而不是处理器调度的对象。处理器调度是严格针对线程进行的，并不考虑被调度线程属于哪个进程。例如，进程 A有10个可运行的线程，进程 B有2个可运行的线程，这12个线程的优先级都相同，则每个线程将得到 1/12的处理器时间。 Windows 2000/XP并不会把处理器时间分成相同的两半，一半给进程 A，另一半给进程B。

为了理解线程调度算法，我们首先要说明 Windows 2000/XP所使用的优先级。

### 3.9.2　Win32中与线程调度相关的应用程序编程接口

表3-6中给出了Win32 API中与线程调度相关的函数列表。更详细的信息可参见 Win32 API的参考文档。

表　3-6

| 与线程调度相关的API函数名 | 函 数 功 能 |
|---|---|
| Suspend/ResumeThread | 挂起一个正在运行的线程或激活一个暂停运行的线程 |
| Get/SetPriorityClass | 读取或设置一个进程的基本优先级类型 |
| Get/SetThreadPriority | 读取或设置一个线程相对优先级 (相对进程优先级类型) |

（续）

| 与线程调度相关的API函数名 | 函 数 功 能 |
|---|---|
| Get/SetProcessAffinityMask | 读取或设置一个进程的亲合处理器集合 |
| SetThreadAffinityMask | 设置线程的亲合处理器集合（必须是进程亲合处理器集合的子集），只允许该线程在指定的处理器集合运行 |
| Get/SetThreadPriorityBoost | 读取或设置Windows 2000/XP暂时提升线程优先级状态；只能在可调范围内提升 |
| SetThreadIdealProcessor | 设置一个特定线程的首选处理器；不限制该线程只能在该处理器上运行 |
| Get/SetProcessPriorityBoost | 读取或设置当前进程的缺省优先级提升控制。该功能用于在创建线程时控制线程优先级的暂时提升状态 |
| SwitchToThread | 当前线程放弃一个或多个时间配额的运行 |
| Sleep | 使当前线程等待指定的一段时间（时间单位为毫秒）。0表示放弃该线程的剩余时间配额 |
| SleepEx | 使当前线程进入等待状态，直到I/O处理完成、有一个与该线程相关的APC或经过一段指定的时间 |

### 3.9.3 线程优先级

如图3-27所示，Windows 2000/XP内部使用32个线程优先级，范围从0到31，它们被分成以下三个部分。

1) 16个实时线程优先级（16～31）。

2) 15个可变线程优先级（1～15）。

3) 一个系统线程优先级（0），仅用于对系统中空闲物理页面进行清零的零页线程。

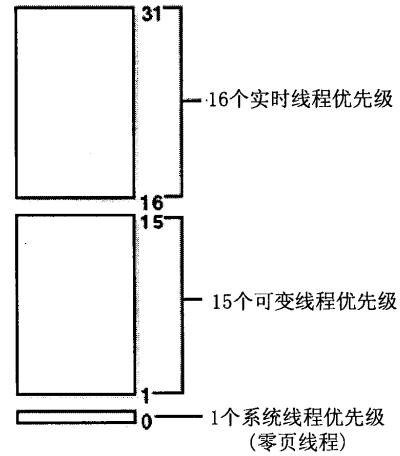


图3-27 线程优先级

线程优先级的指定可从两个不同的角度进行：用户可通过Win32应用程序编程接口来指定线程的优先级，Windows 2000/XP内核也可控制线程的优先级。Win32应用程序编程接口可在进程创建时指定其优先级类型为实时、高级、中上、中级、中下和空闲，并进一步在进程内各线程创建时指定线程的相对优先级为相对实时、相对高级、相对中上、相对中级、相对中下、相对低级和相对空闲。

通过Win32应用程序编程接口指定的线程优先级是由进程优先级类型和线程相对优先级共同控制。图3-28给出Win32线程优先级到Windows 2000/XP内部优先级的映射关系。

图3-28给出的是线程的基本优先级。通过任务管理器（Task Manager）或Win32应用程序编程接口的SetPriorityClass函数可指定进程的基本优先级，线程从继承的进程基本优先级开始运行。

进程基本优先级和线程开始时的优先级通常是缺省地设置为各进程优先级类型的中间值（24、13、10、8、6或4）。Windows 2000/XP的一些系统进程（如会话管理器、服务控制器和本地安全

认证服务器等）的基本优先级比缺省的中级（ 8）要高一些。这样可保证这些进程中的线程在开始时就具有高于缺省值8的优先级。系统进程可使用Windows 2000/XP的内部函数来设置比Win32基本优先级更高的进程基本优先级。

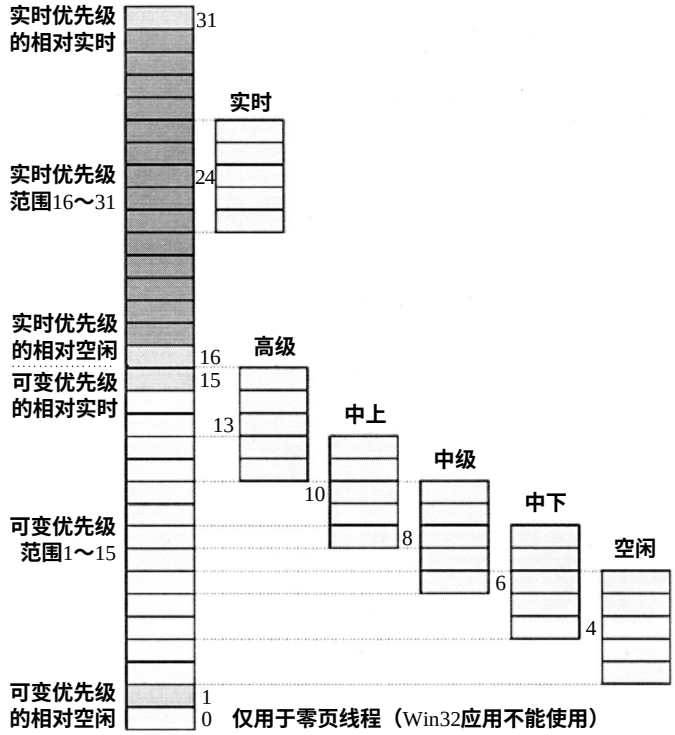


图3-28　Win32和Windows 2000/XP的线程优先级

一个进程仅有单个优先级取值（基本优先级），而一个线程有当前优先级和基本优先级这两个优先级取值。线程的当前优先级可在一定范围（ 1至15）内动态变化，通常会比基本优先级高。Windows 2000/XP从不调整在实时范围（16至31）内的线程优先级，因而这些线程的基本优先级和当前优先级总是一样的。

1. 实时优先级

在应用程序中，用户可在一定范围内升高或降低线程优先级。要把线程的优先级提升到实时优先级，用户必须有升高线程优先级的权限。如果用户试图提升进程优先级到实时类型，但没有相应权限，则相应操作并不会失败，而是提升到高级类型。

我们知道，Windows 2000/XP有许多重要内核系统线程是运行在实时优先级的。如果用户进程在实时优先级运行时间过多，它将可能阻塞关键系统功能（如存储管理器、缓存管理器、本地和网络文件系统、甚至设备驱动程序等 )的执行，阻塞系统线程的运行；但由于硬件中断的优先级比任何线程都要高，因此它不会阻塞硬件中断处理。

在被其他线程抢先时，具有实时优先级的线程的行为与具有可变优先级的线程的行为是不同

的。具有实时优先级的线程在被抢先时，它的时间配额将会被重置成进入运行状态时的初值。

注：虽然Windows 2000/XP有一组优先级称为实时优先级，但是它们并不是通常意义上的实时。Windows 2000/XP并不提供实时操作系统服务，如有保证的中断处理延时或确保线程得到一定执行时间的机制。

2. 中断优先级与线程优先级的关系

如图3-29所示，所有线程都运行在中断优先级0和1。用户态线程运行在中断优先级0，内核态的异步过程调用运行在中断优先级1，它们会中断线程的运行。只有内核态线程可提升自己的中断优先级；虽然高优先级的实时线程可阻塞重要的系统线程执行，但不管用户态线程的优先级是多少，它都不会阻塞硬件中断。

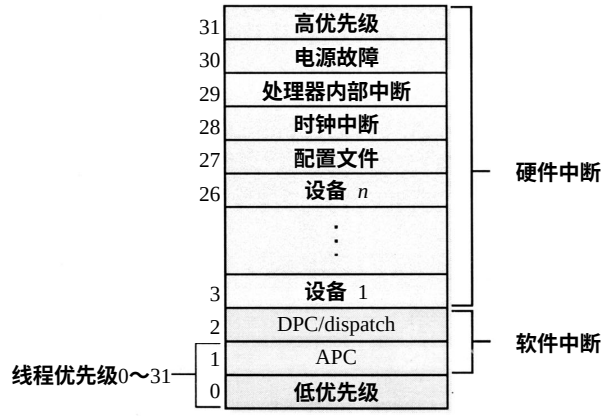


图3-29 中断优先级与线程优先级

线程调度代码是运行在DPC/线程调度中断优先级的。因此，当内核正在选择下一个要运行的线程时，系统中不会有线程正在运行，也不可能修改线程优先级等与调度相关的信息。在多处理器系统中，访问线程调度数据结构是通过请求线程调度器自旋锁(KiDispatcherLock)来实现同步的。

### 3.9.4 线程时间配额

正如前面所述，时间配额是一个线程从进入运行状态到Windows 2000/XP检查是否有其他优先级相同的线程需要开始运行之间的时间总和。一个线程用完了自己的时间配额时，如果没有其他相同优先级的线程，Windows 2000/XP将重新给该线程分配一个新的时间配额，并继续运行。

每个线程都有一个代表本次运行最大时间长度的时间配额。时间配额不是一个时间长度值，而是一个称为配额单位(quantum unit)的整数。

1. 时间配额的计算

缺省时，在Windows 2000专业版中线程开始时的时间配额为6；而在Windows 2000/XP服务器版中线程开始时的时间配额为36。后面我们将介绍如何修改缺省时间配额值。在Windows 2000/XP服务器版中取较长缺省时间配额的原因是要保证客户请求所唤醒的服务器应用有足够的时间在它的时间配额用完前完成客户的请求并回到等待状态。

每次时钟中断，时钟中断服务例程从线程的时间配额中减少一个固定值(3)。如果没有剩余的时间配额，系统将触发时间配额用完处理，选择另外一个线程进入运行状态。在Windows 2000专业版中，由于每个时钟中断时减少的时间配额为3，因此一个线程的缺省运行时间为2个时钟中断间隔；在Windows 2000/XP服务器版中，一个线程的缺省运行时间为12个时钟中断间隔。

如果时钟中断出现时系统正处在DPC/线程调度中断优先级以上(如系统正在执行一个DPC或一个中断服务例程)，当前线程的时间配额仍然要减少。甚至在整个时钟中断间隔期间，当前线

程一条指令也没有执行，它的时间配额在时钟中断中也会被减少。

不同硬件平台的时钟中断间隔是不同的，时钟中断的频率是由硬件抽象层确定的，而不是由内核确定的。例如，大多数x86单处理器系统的时钟中断间隔为10毫秒，大多数x86多处理器系统的时钟中断间隔为15毫秒。

利用Win32的函数GetSystemTimeAdjustment可得到系统的时钟中断间隔。用www.sysinternals.com上的Clockres程序也可得到系统的时钟中断间隔。

在一个空闲的系统上，利用性能监视工具也可大致估计系统的时钟中断间隔。空闲系统是指没有进程在执行I/O操作；通过检查每个进程的I/O计数，可验证系统是否空闲。观察性能监视工具中的处理器对象中的每秒中断次数计数(Interrupts/sec)，该计数平均值的倒数就是系统的时钟中断间隔。

例如，在大多数的x86单处理器系统中，每秒中断次数计数的平均值为100，因此可计算出时钟中断间隔为1/100=0.01秒，即10毫秒。在一个x86多处理器系统中，每秒中断次数计数的平均值为67，则时钟中断间隔为1/67=0.015秒，即15毫秒。

用3个时间配额单位，而一个时间配额单位表示一个时钟中断间隔的目的是，在等待完成时允许减少部分时间配额。当优先级小于14的线程执行一个等待函数(如WaitForSingleObject或WaitForMultipleObjects)时，它的时间配额被减少1个时间配额单位。当优先级大于等于14的线程执行完等待函数后，它的时间配额被重置。

这种部分减少时间配额的做法可解决线程在时钟中断触发前进入等待状态所产生的问题。如果不进行这种部分减少时间配额的操作，一个线程可能永远不会减少它的时间配额。例如，一个线程运行一段时间后进入等待状态，再运行一段时间后又进入等待状态，但在时钟中断出现时它都不是当前线程，则它的时间配额永远也不会因为运行而减少。

2. 时间配额的控制

在系统注册库中的一个注册项“HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation”，允许用户指定线程时间配额的相对长度(长或短)和前台进程的时间配额是否加长。如图3-30所示，该注册项为6位，分成3个字段，每个字段占2位。

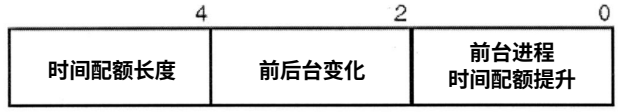


图3-30　注册项Win32PrioritySeparation的定义

时间配额长度字段(Short vs. Long)：1表示长时间配额；2表示短时间配额；0或3表示缺省设置(Windows 2000专业版的缺省设置为短时间配额，Windows 2000服务器版的缺省设置为长时间配额)。

前后台变化字段(Variable vs. Fixed)：1表示改变前台进程时间配额；2表示前后台进程的时间配额相同；0或3表示缺省设置(Windows 2000专业版的缺省设置为改变前台进程时间配额，Windows 2000服务器版的缺省设置为前后台进程的时间配额相同)。

前台进程时间配额提升字段(Foreground Quantum Boost)：该字段的取值只能是0、1或2(取3是非法的，被视为2)。该字段是一个时间配额表索引，用于设置前后台进程的时间配额，后台进程的时间配额为第一项，前台进程的时间配额为第二项。该字段的值保存在内核变量PsPrioritySeparation。

前台进程是指拥有屏幕当前窗口的线程所在的进程。如果当前窗口切换到一个优先级高于空闲优先级类的进程中的某个线程，Win32子系统将用注册项Win32PrioritySeparation的前台进程时间配额字段作为索引，依据一个有3个元素的数组PspForegroundQuantum中的取值，来设置该进程中所有线程的时间配额。该数组的内容由注册项Win32PrioritySeparation的另外2个字段确定。表3-7给出了数组PspForegroundQuantum的可能时间配额设置。

表 3-7

| | 短时间配额 | | | 长时间配额 | | |
|---|---|---|---|---|---|---|
| 改变前台进程时间配额 | 6 | 12 | 18 | 12 | 24 | 36 |
| 前后台进程的时间配额相同 | 18 | 18 | 18 | 36 | 36 | 36 |

下面我们举例说明，为什么在Windows 2000/XP中要增加前台线程的时间配额，而不是提高前台线程的优先级。基于提高前台线程优先级的做法存在一个潜在的问题。假设用户首先启动了一个运行时间很长的电子表格计算程序，然后切换到一个计算密集型的应用(如一个需要复杂图形显示的游戏)。如果前台的游戏进程提高它的优先级，后台的电子表格将会几乎得不到处理器时间。但增加游戏进程的时间配额，则不会停止电子表格计算的执行，而只是给游戏进程的处理器时间多一些。如果用户希望运行一个交互式应用程序时的优先级比其他交互进程的优先级高，可利用任务管理器将进程的优先级类型修改为中上或高级，也可利用命令行在启动应用时使用命令“start /abovenormal”或“start /high”来设置进程优先级类型。在任务管理器中修改进程优先级类型的方法为，在“进程”栏中用鼠标右键激活下拉菜单中的“设置优先级”。

通过直接修改注册项Win32PrioritySeparation来设置时间配额时，用户可对3个字段中的内容进行任意组合。通过控制面板中的性能设置(Performance Options)窗口来设置时间配额时，你只有2种选择。用户可从“控制面板”中的“系统”或“我的电脑”中的“属性”里找到“性能”设置窗口。

为了优化应用的性能，时间配额的设置可为短时间配额和改变前台进程的时间配额，这是Windows 2000专业版的缺省设置。为了优化后台服务的性能，设置可为长时间配额和前后台进程的时间配额相同，这是Windows 2000服务器版的缺省设置。如果在Windows 2000高级服务器或Windows 2000数据中心服务器上安装远程终端服务(Terminal Services)，并且配置该服务器为应用服务器时，时间配额设置为优化应用的性能。

### 3.9.5 调度数据结构

如图3-31所示，为了进行线程调度，内核维护了一组称为“调度器数据结构”的数据结构。调度器数据结构负责记录各线程的状态，如哪些线程处于等待状态、处理器正在执行哪个线程等。

调度器数据结构中的最主要内容是调度器的就绪队列(KiDispatcherReadyListHead)，该队列由一组子队列组成，每个调度优先级有一个队列，其中包括该优先级的等待调度执行的就绪线程。

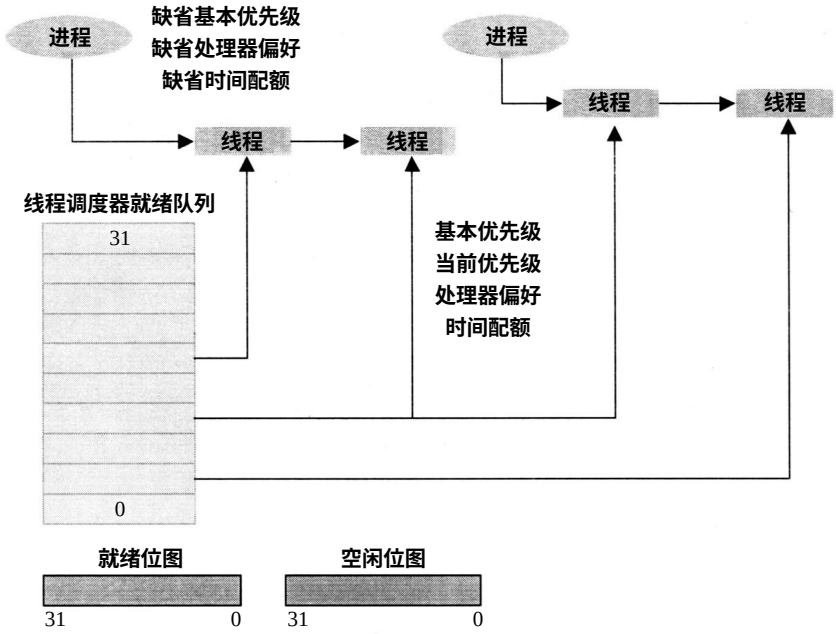


图3-31　线程调度器数据结构

为了提高调度速度，Windows 2000/XP维护了一个称为就绪位图(KiReadySummary)的32位量。就绪位图中的每一位指示一个调度优先级的就绪队列中是否有线程等待运行。 B0与调度优先级0相对应， B1与调度优先级 1相对应，等待。 Windows 2000/XP还维护一个称为空闲位图(KiIdleSummary)的32位量。空闲位图中的每一位指示一个处理器是否处于空闲状态。

如前所述，为了防止调度器代码与线程在访问调度器数据结构时发生冲突，线程调度仅出现在DPC/线程调度中断优先级。但在多处理器系统中，修改调度器数据结构需要额外的步骤来得到内核调度器自旋锁(KiDispatcherLock)，以协调各处理器对调度器数据结构的访问。表 3-8给出了与线程调度相关的内核变量。

表　3-8

| 变 量 名 | 变 量 类 型 | 功 能 说 明 |
|---|---|---|
| KiDispatcherLock | 自旋锁 | 调度器自旋锁 |
| KeNumberProcessors | 字节 | 系统中的可用处理器数目 |
| KeActiveProcessors | 32位位图 | 描述系统中各处理器是否正处于运行状态 |
| KiIdleSummary | 32位位图 | 描述系统中各处理器是否处于空闲状态 |
| KiReadySummary | 32位位图 | 描述各调度优先级是否有就绪线程等待调度 |
| KiDispatcherReadyListHead | 有32个元素的数组 | 32个元素分别指向32个就绪队列 |

### 3.9.6 调度策略

Windows 2000/XP严格基于线程的优先级来确定哪一个线程将占用处理器并进入运行状态。但在实际系统中是如何实现的？下面章节将说明如何基于线程实现优先级驱动的抢先式多任务。需要说明的是，Windows 2000/XP在单处理器系统和多处理器系统中的线程调度是不同的。这里我们首先介绍单处理器系统中的线程调度。

1. 主动切换

首先一个线程可能因为进入等待状态而主动放弃处理器的使用。许多 Win32等待函数调用(如WaitForSingleObject或WaitForMultipleObjects等)都使线程等待某个对象，等待的对象可能有事件、互斥信号量、资源信号量、 I/O操作、进程、线程、窗口消息等。

主动切换可比喻成一个线程在快餐柜台买了一份还未完成的汉堡包。为了不阻塞它后面的就餐者购买快餐，它可以先站在一边等待，以便下一个线程在它等待的时候可以购买 (运行它的例程)。当该线程等待的汉堡包做好时，它会排到相应优先级的就绪队列尾。但读者在后面可以见到，大多数的等待操作都会导致临时性的优先级提高，以便让等待线程可以得到它买的汉堡包，并开始就餐。

图3-32说明了在一个线程进入等待状态时 Windows 2000/XP如何选择一个新线程开始运行。

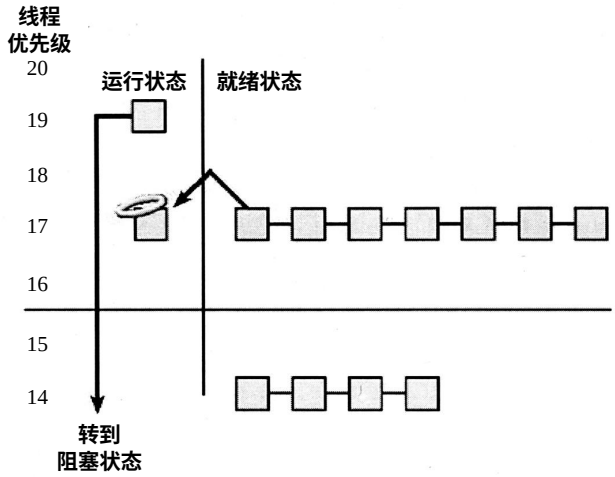


图3-32 主动切换

在图3-32中，正方形表示线程，最上方的线程主动放弃处理器占用，就绪队列中的第一个线程(带光环的正方形)进入运行状态。虽然图 3-32中主动放弃处理器的线程被降低了优先级，但这并不是必须的，可以仅仅是被放入等待对象的等待队列中。如何处理线程的剩余时间配额？通常进入等待状态线程的时间配额不会被重置，而是在等待事件出现时，线程的时间配额被减 1，相当于1/3个时钟间隔；如果线程的优先级大于等于 14，在等待事件出现时，线程的优先级被重置。

2. 抢先

在这种情况下，当一个高优先级线程进入就绪状态时，正在处于运行状态的低优先级线程被

抢先。可能在以下两种情况下出现抢先：

1) 高优先级线程的等待完成，即一个线程等待的事件出现。

2) 一个线程的优先级被增加或减少。

在这两种情况下，Windows 2000/XP都要确定是否让当前线程继续运行或是否当前线程要被一个高优先级线程抢先。

注：用户态下运行的线程可以抢先内核态下运行的线程。在判断一个线程是否被抢先时，并不考虑线程处于用户态还是内核态，调度器只是依据线程优先级进行判断。

当线程被抢先时，它被放回相应优先级的就绪队列的队首。处于实时优先级的线程在被抢先时，时间配额被重置为一个完整的时间片；而处于动态优先级的线程在被抢先时，时间配额不变，重新得到处理器使用权后将运行到剩余的时间配额用完。图 3-33说明了线程抢先的过程。

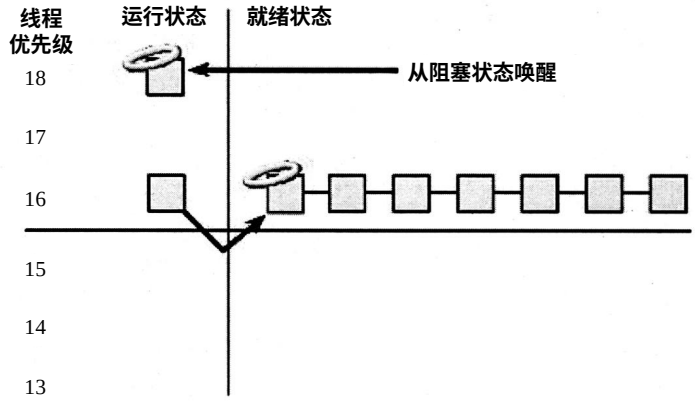


图3-33　线程的抢先调度

在图3-33中，一个优先级为18的线程从等待状态返回并收复处理器，这导致优先级为 16的正在运行的线程被弹回到就绪队列的队首。注意，被抢先的线程是排在就绪队列的队首，而不是队尾。当抢先线程完成运行后，被抢先的线程可继续它的剩余时间配额。在这个例子中，线程的优先级都在实时优先级范围，它们的优先级不会被动态提升。

如果我们把主动切换比喻成一个线程在它等待自己的汉堡包时允许排在它后面的线程可以买快餐，抢先则可比喻成由于美国总统来到快餐店要求买快餐，一个正在运行的线程被挤回到就绪队列。被抢先的线程并不是排到就绪队列的队尾，而只是在总统买快餐时站在一旁；一旦总统离开，它会恢复运行，完成快餐采购。

3. 时间配额用完

当一个处于运行状态的线程用完它的时间配额时， Windows 2000/XP首先必须确定是否需要降低该线程的优先级，然后确定是否需要调度另一个线程进入运行状态。

如果刚用完时间配额的线程的优先级被降低了， Windows 2000/XP将寻找一个更适合的线程进入运行状态；所谓更适合的线程是指优先级高于刚用完时间配额的线程的新设置值的就绪线程。如果刚用完时间配额的线程的优先级没有降低，并且有其他优先级相同的就绪线程，

Windows 2000/XP将选择相同优先级的就绪队列中的下一个线程进入运行状态，刚用完时间配额的线程被排到就绪队列的队尾（即分配一个新的时间配额并把线程状态从运行状态改为就绪状态）。图3-34说明了这个调度过程。如果没有优先级相同的就绪线程可运行，刚用完时间配额的线程将得到一个新的时间配额并继续运行。

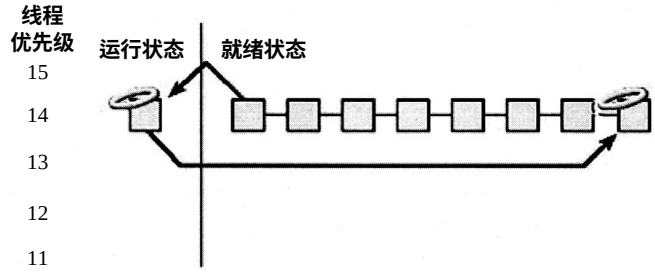


图3-34　时间配额用完时的线程调度

4. 结束

当线程完成运行时，它的状态从运行状态转到终止状态。线程完成运行的原因可能是通过调用ExitThread而从主函数中返回或被其他线程通过调用 TerminateThread来终止。如果处于终止状态的线程对象上没有未关闭的句柄，则该线程将被从进程的线程列表中删除，相关数据结构将被释放。

### 3.9.7　线程优先级提升

在下列5种情况下，Windows 2000/XP会提升线程的当前优先级：

1) I/O操作完成。

2) 信号量或事件等待结束。

3) 前台进程中的线程完成一个等待操作。

4) 由于窗口活动而唤醒图形用户接口线程。

5) 线程处于就绪状态超过一定时间，但没能进入运行状态（处理器饥饿）。

其中，前两条是针对所有线程进行的优先级提升，而后三条是针对某些特殊的线程在正常的优先级提升基础上进行额外的优先级提升。线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。与任何调度算法一样，线程优先级提升也不是完美的，它并不会使所有应用都受益。

注：Windows 2000/XP永远不会提升实时优先级范围内（16至31）的线程优先级。因此，在实时优先级范围内的线程调度总是可以预测的。在使用实时优先级时，Windows 2000/XP假定用户完全了解线程的行为。

1. I/O操作完成后的线程优先级提升

在完成I/O操作后，Windows 2000/XP将临时提升等待该操作线程的优先级，以保证等待 I/O操作的线程能有更多的机会立即开始处理得到的结果。如前所述，为了避免 I/O操作导致对某些

线程的不公平偏好，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1。虽然在DDK头文件中有关于优先级提升的建议值，但线程优先级的实际提升值是由设备驱动程序决定的。与I/O操作相关的线程优先级提升建议值在文件“ Wdm.h ”或“ Ntddk.h ”中以“ #define IO ”串开始处；表3-9是线程优先级提升建议值的列表。设备驱动程序在完成 I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度。

注：在表3-9中，线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大。

表 3-9

| 设　备 | 优先级提升幅度 |
|---|---|
| 磁盘、光驱、并口、视频 | 1 |
| 网络、邮件槽、命名管道、串口 | 2 |
| 键盘、鼠标 | 6 |
| 音频 | 8 |

线程优先级提升是以线程的基本优先级为基点的，不是以线程的当前优先级为基点。如图3-35所示，线程优先级提升后将在提升后的优先级上运行一个时间配额。当用完它的一个时间配额后，线程会降低一个优先级，并运行另一个时间配额。这个降低过程会一直进行下去，直到线程的优先级降低至原来的基本优先级。其他优先级较高的线程仍可抢先因I/O操作而提升了优先级的线程，但被抢先的线程要在提升后的优先级上用完它的时间配额后才降低一个优先级。

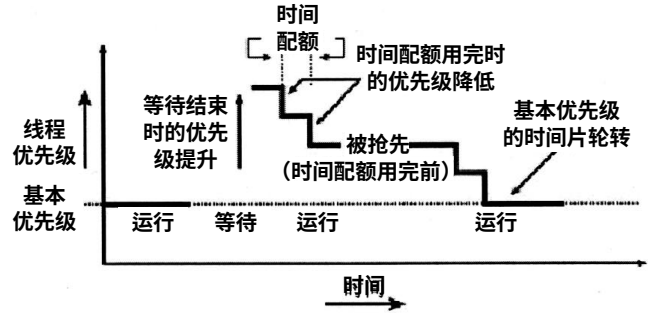


图3-35　线程优先级的提升和降低

如前所述，优先级提升策略仅适用于可变优先级范围（0到15)内的线程。不管线程的优先级提升幅度有多大，提升后的优先级都不会超过15而进入实时优先级。也就是说，一个优先级为14的线程要提升5个优先级，则提升后的线程优先级为15；一个优先级为15的线程提升优先级后的线程优先级还是15。

2. 等待事件和信号量后的线程优先级提升

当一个等待执行事件对象或信号量对象的线程完成等待后，它的线程将提升一个优先级。SetEvent、PulseEvent或ReleaseSemaphore函数调用可导致事件对象或信号量对象等待的结束。

在DDK头文件中定义的常量EVENT_INCREMENT和SEMAPHORE_INCREMENT表示事件和信号量等待结束后的优先级提升幅度。线程在等待事件或信号量后提升优先级的原因与等待I/O操作后的优先级提升的原因类似。阻塞于事件或信号量的线程得到的处理器时间比处理器繁忙型线程要少，这种提升可减少这种不平衡带来的影响。

等待事件或信号量后的线程优先级提升与I/O操作完成时的提升相同：提升是以线程的基本优先级为基点的，而不是以线程的当前优先级为基点的。提升后的优先级永远不会超过15。在等待结束时，线程的时间配额被减1，并在提升后的优先级上执行完剩余的时间配额；随后降低1个优先级，运行一个新的时间配额，直到优先级降低到初始的基本优先级。

3. 前台线程在等待结束后的优先级提升

对于前台进程中的线程，一个内核对象上的等待操作完成时，内核函数KiUnwaitThread会提升线程的当前优先级(不是线程的基本优先级)，提升幅度为变量PsPrioritySeparation的值。窗口子系统负责确定哪一个进程是前台进程。正如有关时间配额的讨论中的描述，变量PsPrioritySeparation是时间配额表的索引，用于选择前台进程中线程的时间配额。

这种针对前台线程的优先级提升是为了改进交互性应用的响应时间特征。在前台应用完成它的等待操作时小幅提升它的优先级，以使它更有可能马上进入运行状态。特别是后台有多个优先级相同的进程时，这种做法可有效改进前台应用的响应时间特征。

与其他类型的优先级提升不同，这种做法在Windows 2000专业版和Windows 2000服务器版中都有效；而且用户不能禁止这种优先级提升，甚至是在用户已利用Win32的函数SetThreadPriorityBoost禁止了其他的优先级提升策略时，也是如此。

4. 图形用户接口线程被唤醒后的优先级提升

拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升。窗口系统(Win32k.sys)在调用函数KeSetEvent时实施这种优先级提升，KeSetEvent函数调用设置一个事件，用于唤醒一个图形用户接口线程。与前一种线程优先级提升类似，这种优先级提升的原因是改进交互应用的响应时间。

5. 对处理器饥饿线程的优先级提升

想象如下情形：一个优先级为7的线程正处于运行状态，另一个优先级为4的线程在这种情况下是不会得到处理器使用权的。但如果一个优先级为11的线程正等待某种被优先级为4的线程锁定的资源，这个优先级为4的线程将永远得不到足够长的处理器时间来完成它的工作，并释放阻塞优先级为11的线程所需要的资源。Windows 2000/XP将如何处理这种情形？在有关虚拟存储的讨论中介绍了一个称为平衡集管理器(balance set manager)的用于内存管理的系统线程，它会每秒钟检查一次就绪队列，看一看是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程。依据系统的时钟中断间隔的不同，300个时钟中断间隔的时间大约为3到4秒。如果找到这个线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额；当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级。如果在该线程结束前出现其他高优先级的就绪线程，该线程会被放回就绪队列，并在就绪队列中超过另外300个时钟中断间隔后再次被提升优先级。

在每次运行时，平衡集管理器并不真正扫描所有就绪线程。为了减少它的处理器占用时间，平衡集管理器只扫描16个就绪线程。如果就绪队列中有更多的线程，它将记住暂停时的位置，并在下一次开始时从当前位置开始扫描。与此同时，平衡集管理器在每次扫描时最多提升 10个线程的优先级。如果在一次扫描中已提升了 10个线程的优先级(这表明系统处于特别繁忙的状态 )，平衡集管理器会停止本次扫描，并在下一次开始时从当前位置开始扫描。

这种算法并不能解决所有优先级倒置的问题，但它很有效。处于处理器饥饿的线程通常都能得到足够的处理器时间来完成它处理的操作并重新进入等待状态。

### 3.9.8　对称多处理器系统上的线程调度

如果完全基于线程优先级进行线程调度，在多处理器系统中会出现什么情况？当 Windows 2000/XP试图调度优先级最高的可执行线程时，有几个因素会影响到处理器的选择。 Windows 2000/XP只保证一个优先级最高的线程处于运行状态。在描述算法前，我们首先定义几个术语。

1. 亲合关系

每个线程都有一个亲合掩码，描述该线程可在哪些处理器上运行。线程的亲合掩码是从进程的亲合掩码继承得到的。缺省时，所有进程 (即所有线程)的亲合掩码为系统中所有可用处理器的集合。也就是说，所有线程可在所有处理器上运行。应用程序通过调用 SetProcessAffinityMask或SetThreadAffinityMask函数来修改缺省的亲合掩码。

2. 线程的首选处理器和第二处理器

每个线程在对应的内核线程控制块中都保存有两个处理器标识：

1) 首选处理器：线程运行时的偏好处理器。

2) 第二处理器：线程第二个选择的运行处理器。

线程的首选处理器是基于进程控制块的索引值在线程创建时随机选择的。索引值在每个线程创建时递增，这样进程中每个新线程得到的首选处理器会在系统中的可用处理器中循环。线程创建后，Windows 2000/XP系统不会修改线程的首选处理器设置；但应用程序可通过SetThreadIdealProcessor函数来修改线程的首选处理器。

3. 就绪线程的运行处理器选择

当线程进入运行状态时， Windows 2000/XP首先试图调度该线程到一个空闲处理器上运行。如果有多个空闲处理器，线程调度器的调度顺序为：首先是线程的首选处理器，其次是线程的第二处理器，第三是当前执行处理器 (即正在执行调度器代码的处理器 )。如果这些处理器都不是空闲的，Windows 2000/XP将依据处理器标识从高到低扫描系统中的空闲处理器状态，选择找到的第一个空闲处理器。

如果线程进入就绪状态时所有处理器都处于繁忙状态， Windows 2000/XP将检查它是否可抢先一个处于运行状态或备用状态的线程。检查的顺序如下：首先是线程的首选处理器，其次是线程的第二处理器。如果这两个处理器都不在线程的亲合掩码中， Windows 2000/XP将依据活动处理器掩码选择该线程可运行的编号最大的处理器。注：线程的亲合掩码与首选处理器、第二处理器的设置是相互独立的，首选和第二处理器由系统在创建线程时指定，而亲合掩码由用户选择。

如果被选中的处理器已有一个线程处于备用状态（即下一个在该处理器上运行的线程），并且该线程的优先级低于正在检查的线程，则正在检查的线程取代原处于备用状态的线程，成为该处理器的下一个运行线程。如果已有一个线程正在被选中的处理器上运行，Windows 2000/XP将检查当前运行线程的优先级是否低于正在检查的线程；如果正在检查的线程优先级高，则标记当前运行线程为被抢先，系统会发出一个处理器间中断，以抢先正在运行的线程，让新线程在该处理器上运行。

注：Windows 2000/XP并不检查所有处理器上的运行线程和备用线程的优先级，而仅仅按上述过程检查一个被选中处理器上的运行线程和备用线程的优先级。如果在被选中的处理器上没有线程可被抢先，则新线程放入相应优先级的就绪队列，并等待调度执行。

4. 为特定的处理器调度线程

在有些情况（如线程降低它的优先级、修改它的亲合处理器、推迟或放弃执行等）下，Windows 2000/XP必须选择一个新线程，在刚让出的处理器上运行。在单处理器系统中，Windows 2000/XP简单地从就绪队列中选择最高优先级的第一个线程。在多处理器系统，Windows 2000/XP不能简单地从就绪队列中取第一个线程，它要寻找一个满足下列四个条件之一的线程。

1) 线程的上一次运行是在该处理器上。

2) 线程的首选处理器是该处理器。

3) 处于就绪状态的时间超过2个时间配额单位。

4) 优先级大于等于24。

显然，由线程亲合掩码限制不能在指定处理器上运行的线程将在检查中跳过。如果Windows 2000/XP不能找到满足要求的线程，它将从就绪队列的队首取第一个线程进入运行状态。

为什么在为处理器选择运行线程时要考虑线程上一次运行时使用的处理器？主要的原因是速度问题。如果线程的连续两次运行都在同一个处理器上，将增加线程数据仍保留在处理器第二级缓存的可能性。

5. 最高优先级就绪线程可能不处于运行状态

如前所述，在多处理器系统中，Windows 2000/XP并不总是选择优先级最高的线程在一个指定的处理器上运行。于是有可能出现这种情况，一个比当前正在运行的线程优先级更高的线程处于就绪状态，但不能立即抢先当前线程，进入运行状态。

一种高优先级线程可能不抢先当前线程的情况是，线程的亲合掩码限制线程只能在一部分可用处理器上运行。在这种情况下，某线程可运行的处理器上运行着高优先级线程，而其他处理器可能空闲或运行着该线程可抢先的低优先级线程。虽然把一个处于运行状态的线程从一个处理器移到另一个处理器的做法可允许另一个由于亲合掩码限制而不能运行的线程进入运行状态，但Windows 2000/XP不会因此把一个正在运行的线程从一个处理器移到另一个处理器上。

例如，假设0号处理器上正运行着一个可在任何处理器上运行的优先级为8的线程，1号处理器上正运行着一个可在任何处理器上运行的优先级为4的线程；这时一个只能在0号处理器上运行的优先级为6的线程进入就绪状态。在这种情况下，优先级为6的线程只能等待0号处理器上优先

级为8的线程结束。因为Windows 2000/XP不会为了让优先级为6的线程在0号处理器上运行，而把优先级为8的线程从0号处理器移到1号处理器。即0号处理器上的优先级为8的线程不会抢先1号处理器上优先级为4的线程。

### 3.9.9　空闲线程

如果在一个处理器上没有可运行的线程，Windows 2000/XP会调度相应处理器上对应的空闲线程。因为在多处理器系统中可能两个处理器同时运行空闲线程，所以系统中的每个处理器都有一个对应的空闲线程。Windows 2000/XP给空闲线程指定的线程优先级为0，但实际上该空闲线程只在没有其他线程要运行时才运行。空闲线程的功能就是在一个循环中检测是否有要进行的工作。虽然不同处理器结构下空闲线程的流程有一些区别，但基本的控制流程都是如下所描述的过程。

1) 处理所有待处理的中断请求。
2) 检查是否有待处理的DPC请求。如果有，则清除相应软中断并执行DPC。
3) 检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态。
4) 调用硬件抽象层的处理器空闲例程，执行相应的电源管理功能。

在Windows 2000/XP下有多种进程浏览工具，不同浏览工具给出的空闲线程的名字会不同，如系统空闲进程、空闲进程、系统进程等。

## 习题

3.1　试说明进程与程序的关系。
3.2　试比较进程与线程的区别。
3.3　在进程模型中引入挂起状态的目的是什么？
3.4　试分析内核线程、用户线程与轻量级进程的不同点。
3.5　试说明进程模型中等待状态与阻塞状态的区别。
3.6　写一个程序，通过递归创建进程1000次来测定Windows 2000中的进程创建速度。
3.7　写一个程序，通过递归创建线程1000次来测定Windows 2000中的线程创建速度。
3.8　写一个程序，通过创建新进程来执行一个命令列表。
3.9　写一个程序，通过创建新线程来执行一个命令列表。
3.10　写一个程序，每过2秒钟显示一行提示，一共显示10行提示信息。要求在前5行提示时，禁止用Ctrl-C来终止程序的执行；而在后5行提示时，可用Ctrl-C来终止程序的执行。
3.11　完成程序A，实现在创建后等待定长时间k后终止，k为命令行参数。完成程序B，在后台创建n个进程，执行程序A(命令行参数为1至2k间的一个随机数)；随后等待定长时间k后结束。观察整个过程中程序B和它所创建的n个进程的关系。
3.12　分别使用消息、管道机制来实现三个进程间的数据传递。第一个进程从数据文件中读入数据，并按字节进行变换T后传递给第二个进程。第二进程对收到的数据按字节进行变换T后传递给第三个进程。第三个进程对收到的数据按字节进行变换T后，把结果存入文件。变换T可为加1后模256。

3.13 **试说明**Windows 2000**中的中断优先级与线程优先级的关系。**

3.14 **试说明**Windows 2000**中的线程优先级控制机制。**

3.15 **试说明在什么情况下线程的时间配额会被增加。**

3.16 Windows 2000**专业版和服务器版的缺省时间配额是如何得到的？**

3.17 **试说明**Windows 2000**的线程调度算法是哪些处理器调度算法的综合。**

3.18 **为什么在**Windows 2000**中完成**I/O**操作后的线程优先级提升是以线程的基本优先级为基准点，而前台线程在等待结束后的优先级提升是以线程的当前优先级为基准点？**

3.19 **试说明**Windows 2000**在多处理器系统中的线程调度过程。**

3.20 **为什么**Windows 2000**在多处理器系统中不能保证各处理器上正在运行线程的优先级都不低于就绪线程的最高优先级？**

## 参考文献

1. David A. Solomon, Mark E. Russinovich. Inside Microsoft Windows 2000. 3rd Edition. Microsoft Press, 2000

2. Gary Nutt. Operating System Project Using Windows NT. Addison-Wesley

3. The July 2000 release of the MSDN Library

4. William Stallings. Operating Systems.3rd edition. **北京：清华大学出版社**, 1998

5. **张尧学，史美林编**. **计算机操作系统教程**. **北京：清华大学出版社，** 1993

6. Uresh Vahalia. UNIX**高级教程 —— 系统技术内幕**. **北京：清华大学出版社，** 1999