

## 第7章 线程的调度、优先级和亲缘性

抢占式操作系统必须使用某种算法来确定哪些线程应该在何时调度和运行多长时间。本章将要介绍Microsoft Windows 98和Windows 2000使用的一些算法。

上一章介绍了每个线程是如何拥有一个上下文结构的,这个结构维护在线程的内核对象中。这个上下文结构反映了线程上次运行时该线程的CPU寄存器的状态。每隔20ms左右,Windows要查看当前存在的所有线程内核对象。在这些对象中,只有某些对象被视为可以调度的对象。Windows选择可调度的线程内核对象中的一个,将它加载到CPU的寄存器中,它的值是上次保存在线程的环境中的值。这项操作称为上下文转换。Windows实际上保存了一个记录,它说明每个线程获得了多少个运行机会。使用Microsoft Spy++这个工具,就可以了解这个情况。图7-1显示了一个线程的属性。注意,该线程已经被调度了37 379次。

目前,线程正在执行代码,并对它的进程的地址空间中的数据进行操作。再过20ms左右,Windows就将CPU的寄存器重新保存到线程的上下文中。线程不再运行。系统再次查看其余的可调度线程内核对象,选定另一个线程的内核对象,将该线程的上下文加载到CPU的寄存器中,然后继续运行。当系统引导时,便开始加载线程的上下文,让线程运行,保存上下文和重复这些操作,直到系统关闭。

总之,这就是系统对线程进行调度的过程。这很简单,是不是? Windows被称为抢占式多线程操作系统,因为一个线程可以随时停止运行,随后另一个线程可进行调度。如你所见,可以对它进行一定程度的控制,但是不能太多。记住,无法保证线程总是能够运行,也不能保证线程能够得到整个进程,无法保证其他线程不被允许运行等等。

**注意** 程序员常常问我,如何才能保证线程在某个事件的某个时间段内开始运行,比如,如何才能确保某个线程在数据从串行端口传送过来的1ms内开始运行呢?我的回答是,办不到。实时操作系统才能作出这样的承诺,但是Windows不是实时操作系统。实时操作系统必须清楚地知道它是在什么硬件上运行,这样它才能知道它的硬盘控制器和键盘等的等待时间。Microsoft对Windows规定的目标是,使它能够在各种不同的硬件上运行,即能够在不同的CPU、不同的驱动器和不同的网络上运行。简而言之,Windows没有设计成为一种实时操作系统。

尽管应强调这样一个概念,即系统只调度可以调度的线程,但是实际情况是,系统中的大多数线程是不可调度的线程。例如,有些线程对象的暂停计数大于1。这意味着该线程已经暂停运行,不应该给它安排任何CPU时间。通过调用使用CREATE\_SUSPENDED标志的CreateProcess或CreateThread函数,可以创建一个暂停的线程。(本章后面还要介绍Suspend

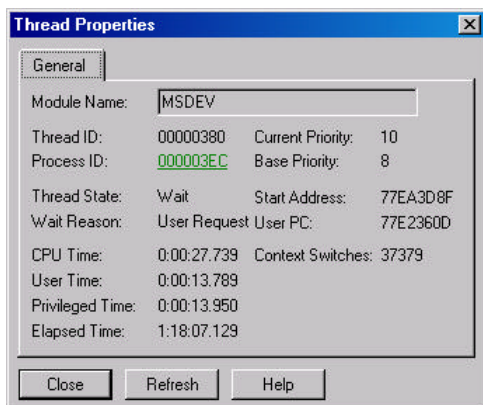


图7-1 线程的属性

Thread和ResumeThread函数。)

除了暂停的线程外，其他许多线程也是不可调度的线程，因为它们正在等待某些事情的发生。例如，如果运行Notepad，但是并不键入任何数据，那么Notepad的线程就没有什么事情要做。系统不给无事可做的线程分配CPU时间。当移动Notepad的窗口时，或者Notepad的窗口需要刷新它的内容，或者将数据键入Notepad，系统就会自动使Notepad的线程成为可调度的线程。这并不意味着Notepad的线程立即获得了CPU时间。它只是表示Notepad的线程有事情可做，系统将设法在某个时间（不久的将来）对它进行调度。

## 7.1 暂停和恢复线程的运行

在线程内核对象的内部有一个值，用于指明线程的暂停计数。当调用CreateProcess或CreateThread函数时，就创建了线程的内核对象，并且它的暂停计数被初始化为1。这可以防止线程被调度到CPU中。当然，这是很有用的，因为线程的初始化需要时间，你不希望在系统做好充分的准备之前就开始执行线程。

当线程完全初始化好了之后，CreateProcess或CreateThread要查看是否已经传递了CREATE\_SUSPENDED标志。如果已经传递了这个标志，那么这些函数就返回，同时新线程处于暂停状态。如果尚未传递该标志，那么该函数将线程的暂停计数递减为0。当线程的暂停计数是0的时候，除非线程正在等待其他某种事情的发生，否则该线程就处于可调度状态。

在暂停状态中创建一个线程，就能够在线程有机会执行任何代码之前改变线程的运行环境（如优先级）。一旦改变了线程的环境，必须使线程成为可调度线程。要进行这项操作，可以调用ResumeThread，将调用CreateThread函数时返回的线程句柄传递给它（或者是将传递给CreateProcess的ppiProcInfo参数指向的线程句柄传递给它）：

```
DWORD ResumeThread(HANDLE hThread);
```

如果ResumeThread函数运行成功，它将返回线程的前一个暂停计数，否则返回0xFFFFFFFF。

单个线程可以暂停若干次。如果一个线程暂停了3次，它必须恢复3次，然后它才可以被分配给一个CPU。当创建线程时，除了使用CREATE\_SUSPENDED外，也可以调用SuspendThread函数来暂停线程的运行：

```
DWORD SuspendThread(HANDLE hThread);
```

任何线程都可以调用该函数来暂停另一个线程的运行（只要拥有线程的句柄）。不用说，线程可以自行暂停运行，但是不能自行恢复运行。与ResumeThread一样，SuspendThread返回的是线程的前一个暂停计数。线程暂停的最多次数是MAXIMUM\_SUSPEND\_COUNT次（在WinNT.h中定义为127）。注意，SuspendThread与内核方式的执行是异步进行的，但是在线程恢复运行之前，不会发生用户方式的执行。

在实际环境中，调用SuspendThread时必须小心，因为不知道暂停线程运行时它在进行什么操作。如果线程试图从堆栈中分配内存，那么该线程将在该堆栈上设置一个锁。当其他线程试图访问该堆栈时，这些线程的访问就被停止，直到第一个线程恢复运行。只有确切知道目标线程是什么（或者目标线程正在做什么），并且采取强有力的措施来避免因暂停线程的运行而带来的问题或死锁状态，SuspendThread才是安全的（死锁和其他线程同步问题将在第8、9和10章介绍）。

## 7.2 暂停和恢复进程的运行

对于Windows来说，不存在暂停或恢复进程的概念，因为进程从来不会被安排获得 CPU 时间。但是，曾经有人无数次问我如何暂停进程中的所有线程的运行。Windows 确实允许一个进程暂停另一个进程中的所有线程的运行，但是从事暂停操作的进程必须是个调试程序。特别是，进程必须调用 WaitForDebugEvent 和 ContinueDebugEvent 之类的函数。

由于竞争的原因，Windows 没有提供其他方法来暂停进程中所有线程的运行。例如，虽然许多线程已经暂停，但是仍然可以创建新线程。从某种意义上说，系统必须在这个时段内暂停所有新线程的运行。Microsoft 已经将这项功能纳入了系统的调试机制。

虽然无法创建绝对完美的 SuspendProcess 函数，但是可以创建一个该函数的实现代码，它能够在许多条件下出色地运行。下面是我的 SuspendProcess 函数的实现代码：

```
VOID SuspendProcess(DWORD dwProcessID, BOOL fSuspend) {

    // Get the list of threads in the system.
    HANDLE hSnapshot = CreateToolhelp32Snapshot(
        TH32CS_SNAPTHREAD, dwProcessID);

    if (hSnapshot != INVALID_HANDLE_VALUE) {

        // Walk the list of threads.
        THREADENTRY32 te = { sizeof(te) };
        BOOL fOk = Thread32First(hSnapshot, &te);
        for (; fOk; fOk = Thread32Next(hSnapshot, &te)) {

            // Is this thread in the desired process?
            if (te.th32OwnerProcessID == dwProcessID) {

                // Attempt to convert the thread ID into a handle.
                HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,
                    FALSE, te.th32ThreadID);

                if (hThread != NULL) {

                    // Suspend or resume the thread.
                    if (fSuspend)
                        SuspendThread(hThread);
                    else
                        ResumeThread(hThread);
                }
                CloseHandle(hThread);
            }
        }
        CloseHandle(hSnapshot);
    }
}
```

我的 SuspendProcess 函数使用 ToolHelp 函数来枚举系统中的线程列表。当我找到作为指定进程的组成部分的线程时，我调用 OpenThread：

```
HANDLE OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadId);
```

这个新 Windows 2000 函数负责找出带有匹配的线程 ID 的线程内核对象，对内核对象的使用计数进行递增，然后返回对象的句柄。运用这个句柄，我调用 `SuspendThread` (或 `ResumeThread`)。由于 `OpenThread` 在 Windows 2000 中是个新函数，因此我的 `SuspendProcess` 函数在 Windows 95 或 Windows 98 上无法运行，在 Windows NT 4.0 或更早的版本上也无法运行。

也许你懂得为什么 `SuspendProcess` 不能总是运行，原因是当枚举线程组时，新线程可以被创建和撤消。因此，当我调用 `CreateToolhelp32Snapshot` 后，一个新线程可能会出现在目标进程中，我的函数将无法暂停这个新线程。过了一些时候，当调用 `SuspendProcess` 函数来恢复线程的运行时，它将恢复它从未暂停的一个线程的运行。更糟糕的是，当枚举线程 ID 时，一个现有的线程可能被撤消，一个新线程可能被创建，这两个线程可能拥有相同的 ID。这将会导致该函数暂停任意些个（也许在目标进程之外的一个进程中的）线程的运行。

当然，这些情况不太可能出现。如果非常了解目标进程是如何运行的，那么这些问题也许根本不是问题。我提供这个函数供酌情使用。

### 7.3 睡眠方式

线程也能告诉系统，它不想在某个时间段内被调度。这是通过调用 `Sleep` 函数来实现的：

```
VOID Sleep(DWORD dwMilliseconds);
```

该函数可使线程暂停自己的运行，直到 `dwMilliseconds` 过去为止。关于 `Sleep` 函数，有下面几个重要问题值得注意：

- 调用 `Sleep`，可使线程自愿放弃它剩余的时间片。
- 系统将在大约的指定毫秒数内使线程不可调度。不错，如果告诉系统，想睡眠 100ms，那么可以睡眠大约这么长时间，但是也可能睡眠数秒钟或者数分钟。记住，Windows 不是个实时操作系统。虽然线程可能在规定的时间被唤醒，但是它能否做到，取决于系统中还有什么操作正在进行。
- 可以调用 `Sleep`，并且为 `dwMilliseconds` 参数传递 `INFINITE`。这将告诉系统永远不要调度该线程。这不是一件值得去做的事情。最好是让线程退出，并还原它的堆栈和内核对象。
- 可以将 0 传递给 `Sleep`。这将告诉系统，调用线程将释放剩余的时间片，并迫使系统调度另一个线程。但是，系统可以对刚刚调用 `Sleep` 的线程重新调度。如果不存在多个拥有相同优先级的可调度线程，就会出现这种情况。

### 7.4 转换到另一个线程

系统提供了一个称为 `SwitchToThread` 的函数，使得另一个可调度线程（如果存在能够运行）：

```
BOOL SwitchToThread();
```

当调用这个函数的时候，系统要查看是否存在一个迫切需要 CPU 时间的线程。如果没有线程迫切需要 CPU 时间，`SwitchToThread` 就会立即返回。如果存在一个迫切需要 CPU 时间的线程，`SwitchToThread` 就对该线程进行调度（该线程的优先级可能低于调用 `SwitchToThread` 的线程）。这个迫切需要 CPU 时间的线程可以运行一个时间段，然后系统调度程序照常运行。

该函数允许一个需要资源的线程强制另一个优先级较低、而目前却拥有该资源的线程放弃该资源。如果调用 `SwitchToThread` 函数时没有其他线程能够运行，那么该函数返回 `FALSE`，否则返回一个非 0 值。

调用SwitchToThread函数与调用Sleep是相似的，并且传递给它一个 0ms的超时。差别是SwitchToThread允许优先级较低的线程运行。即使低优先级线程迫切需要 CPU时间，Sleep也能够立即对调用线程重新进行调度。

Windows 98 Windows 98 没有配备该函数的非常有用的实现代码。

## 7.5 线程的运行时间

有时想要计算线程执行某个任务需要多长的时间。许多人采取的办法是编写类似下面的代码：

```
// Get the current time (start time).
DWORD dwStartTime = GetTickCount();

// Perform complex algorithm here.

// Subtract start time from current time to get duration.
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

这个代码做了一个简单的假设：即它不会被中断。但是，在抢占式操作系统中，永远无法知道线程何时被赋予CPU时间。当取消线程的CPU时间时，就更难计算线程执行不同任务时所用的时间。我们需要一个函数，以便返回线程得到的 CPU时间的数量。幸运的是，Windows提供了一个称为GetThreadTimes的函数，它能返回这些信息：

```
BOOL GetThreadTimes(
    HANDLE hThread,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);
```

GetThreadTimes函数返回4个不同的时间值，这些值如表7-1所示。

表7-1 GetThreadTimes 函数的返回时间值

时 间 值	含 义
创建时间	用英国格林威治时间 1601 年 1 月 1 日午夜后 100ns 的时间间隔表示的英国绝对值，用于指明线程创建的时间
退出时间	用英国格林威治时间 1601 年 1 月 1 日午夜后 100ns 的时间间隔表示的英国绝对值，用于指明线程退出的时间。如果线程仍然在运行，退出时间则未定义
内核时间	一个相对值，用于指明线程执行操作系统代码已经经过了多少个 100ns 的 CPU 时间
用户时间	一个相对值，用于指明线程执行应用程序代码已经经过了多少个 100ns 的 CPU 时间

使用这个函数，可以通过使用下面的代码确定执行复杂的算法时需要的时间量：

```
__int64 FileTimeToQuadWord (PFILETIME pft) {
    return(Int64ShlMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}
```

```
void PerformLongOperation () {
```



```

FILETIME ftKernelTimeStart, ftKernelTimeEnd;
FILETIME ftUserTimeStart, ftUserTimeEnd;
FILETIME ftDummy;
__int64 qwKernelTimeElapsed, qwUserTimeElapsed,
      qwTotalTimeElapsed;

// Get starting times.
GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
      &ftKernelTimeStart, &ftUserTimeStart);

// Perform complex algorithm here.

// Get ending times.
GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
      &ftKernelTimeEnd, &ftUserTimeEnd);

// Get the elapsed kernel and user times by converting the start
// and end times from FILETIMES to quad words, and then subtract
// the start times from the end times.
qwKernelTimeElapsed = FileTimeToQuadWord(&ftKernelTimeEnd) -
      FileTimeToQuadWord(&ftKernelTimeStart);

qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd) -
      FileTimeToQuadWord(&ftUserTimeStart);

// Get total time duration by adding the kernel and user times.
qwTotalTimeElapsed = qwKernelTimeElapsed + qwUserTimeElapsed;

// The total elapsed time is in qwTotalTimeElapsed.
}

```

注意，GetProcessTimes是个类似GetThreadTimes的函数，适用于进程中的所有线程：

```

BOOL GetProcessTimes(
    HANDLE hProcess,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);

```

GetProcessTimes返回的时间适用于某个进程中的所有线程（甚至是已经终止运行的线程）。例如，返回的内核时间是所有进程的线程在内核代码中经过的全部时间的总和。

Windows 98 遗憾的是，GetThreadTimes和GetProcessTimes这两个函数在Windows 98中不起作用。在Windows 98中，没有一个可靠的机制可供应用程序来确定线程或进程已经使用了多少CPU时间。

对于高分辨率的配置文件来说，GetThreadTimes并不完美。Windows确实提供了一些高分辨率性能函数：

```

BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);

BOOL QueryPerformanceCounter(LARGE_INTEGER* pliCount);

```

虽然这些函数认为，正在执行的线程并没有得到抢占的机会，但是高分辨率的配置文件是

为短期存在的代码块设置的。为了使这些函数运行起来更加容易一些，我创建了下面这个 C++ 类：

```
class CStopwatch {
public:
    CStopwatch() { QueryPerformanceFrequency(&m_liPerfFreq); Start();

    void Start() { QueryPerformanceCounter(&m_liPerfStart); }

    __int64 Now() const { //
        Returns # of milliseconds since Start was called
        LARGE_INTEGER liPerfNow;
        QueryPerformanceCounter(&liPerfNow);
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)
            / m_liPerfFreq.QuadPart);
    }

private:
    LARGE_INTEGER m_liPerfFreq; // Counts per second
    LARGE_INTEGER m_liPerfStart; // Starting count
};
```

使用这个类如下：

```
// Create a stopwatch timer (which defaults to the current time).
CStopwatch stopwatch;

// Execute the code I want to profile here.

// Get how much time has elapsed up to now.
__int64 qwElapsedTime = stopwatch.Now();

// qwElapsedTime indicates how long the profiled code
// executed in milliseconds.
```

## 7.6 运用结构环境

现在应该懂得环境结构在线程调度中所起的重要作用了。环境结构使得系统能够记住线程的状态，这样，当下次线程拥有可以运行的 CPU 时，它就能够找到它上次中断运行的地方。

知道这样低层的数据结构也会完整地记录在 Platform SDK 文档中确实使人吃惊。不过如果查看该文档中的 CONTEXT 结构，会看到下面这段文字：

“CONTEXT 结构包含了特定处理器的寄存器数据。系统使用 CONTEXT 结构执行各种内部操作。目前，已经存在为 Intel、MIPS、Alpha 和 PowerPC 处理器定义的 CONTEXT 结构。若要了解这些结构的定义，参见头文件 WinNT.h”。

该文档并没有说明该结构的成员，也没有描述这些成员是谁，因为这些成员要取决于 Windows 2000 在哪个 CPU 上运行。实际上，在 Windows 定义的所有数据结构中，CONTEXT 结构是特定于 CPU 的唯一数据结构。

那么 CONTEXT 结构中究竟存在哪些东西呢？它包含了主机 CPU 上的每个寄存器的数据结构。在 x86 计算机上，数据成员是 Eax、Ebx、Ecx、Edx 等等。如果是 Alpha 处理器，那么数据成员包括 IntV0、IntT0、IntT1、IntS0、IntRa 和 IntZero 等等。下面这个代码段显示了 x86 CPU 的完整的 CONTEXT 结构：

```
typedef struct _CONTEXT {  
  
    //  
    // The flags values within this flag control the contents of  
    // a CONTEXT record.  
    //  
    // If the context record is used as an input parameter, then  
    // for each portion of the context record controlled by a flag  
    // whose value is set, it is assumed that that portion of the  
    // context record contains valid context. If the context record  
    // is being used to modify a threads context, then only that  
    // portion of the threads context will be modified.  
    //  
    // If the context record is used as an IN OUT parameter to capture  
    // the context of a thread, then only those portions of the thread's  
    // context corresponding to set flags will be returned.  
    //  
    // The context record is never used as an OUT only parameter.  
    //  
  
    DWORD ContextFlags;  
  
    //  
    // This section is specified/returned if CONTEXT_DEBUG_REGISTERS is  
    // set in ContextFlags. Note that CONTEXT_DEBUG_REGISTERS is NOT  
    // included in CONTEXT_FULL.  
    //  
  
    DWORD   Dr0;  
    DWORD   Dr1;  
    DWORD   Dr2;  
    DWORD   Dr3;  
    DWORD   Dr6;  
    DWORD   Dr7;  
  
    //  
    // This section is specified/returned if the  
    // ContextFlags word contains the flag CONTEXT_FLOATING_POINT.  
    //  
  
    FLOATING_SAVE_AREA FloatSave;  
  
    //  
    // This section is specified/returned if the  
    // ContextFlags word contains the flag CONTEXT_SEGMENTS.  
    //  
  
    DWORD   SegGs;  
    DWORD   SegFs;  
    DWORD   SegEs;  
    DWORD   SegDs;  
  
    //  
    // This section is specified/returned if the  
    // ContextFlags word contains the flag CONTEXT_INTEGER.
```



```
//
DWORD   Edi;
DWORD   Esi;
DWORD   Ebx;
DWORD   Edx;
DWORD   Ecx;
DWORD   Eax;
//
// This section is specified/returned if the
// ContextFlags word contains the flag CONTEXT_CONTROL.
//

DWORD   Ebp;
DWORD   Eip;
DWORD   SegCs;           // MUST BE SANITIZED
DWORD   EFlags;         // MUST BE SANITIZED
DWORD   Esp;
DWORD   SegSs;

//
// This section is specified/returned if the ContextFlags word
// contains the flag CONTEXT_EXTENDED_REGISTERS.
// The format and contexts are processor specific
//

BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];

} CONTEXT;
```

CONTEXT结构可以分成若干个部分。CONTEXT\_CONTROL包含CPU的控制寄存器，比如指令指针、堆栈指针、标志和函数返回地址（与x86处理器不同，Alpha CPU在调用函数时，将该函数的返回地址放入一个寄存器中）。CONTEXT\_INTEGER用于标识CPU的整数寄存器。CONTEXT\_FLOATING\_POINT用于标识CPU的浮点寄存器。CONTEXT\_SEGMENTS用于标识CPU的段寄存器（仅为x86处理器）。CONTEXT\_DEBUG\_REGISTER用于标识CPU的调试寄存器（仅为x86处理器）。CONTEXT\_EXTENDED\_REGISTERS用于标识CPU的扩展寄存器（仅为x86处理器）。

Windows实际上允许查看线程内核对象的内部情况，以便抓取它当前的一组CPU寄存器。若要进行这项操作，只需要调用GetThreadContext函数：

```
BOOL GetThreadContext(
    HANDLE hThread,
    PCONTEXT pContext);
```

若要调用该函数，只需指定一个CONTEXT结构，对某些标志（该结构的ContextFlags成员）进行初始化，指明想要收回哪些寄存器，并将该结构的地址传递给GetThreadContext。然后该函数将数据填入你要求的成员。

在调用GetThreadContext函数之前，应该调用SuspendThread，否则，线程可能被调度，而且线程的环境可能与你收回的不同。一个线程实际上有两个环境。一个是用户方式，一个是内核方式。GetThreadContext只能返回线程的用户方式环境。如果调用SuspendThread来停止线程的运行，但是该线程目前正在用内核方式运行，那么，即使SuspendThread实际上尚未暂停该

线程的运行，它的用户方式仍然处于稳定状态。线程在恢复用户方式之前，它无法执行更多的用户方式代码，因此可以放心地将线程视为处于暂停状态，GetThreadContext函数将能正常运行。

CONTEXT结构的ContextFlags成员并不与任何CPU寄存器相对应。无论是何种CPU结构，该成员存在于所有CONTEXT结构定义中。ContextFlags成员用于向GetThreadContext函数指明你想检索哪些寄存器。例如，如果想要获得线程的控制寄存器，可以编写下面的代码：

```
// Create a CONTEXT structure.
CONTEXT Context;

// Tell the system that we are interested in only the
// control registers.
Context.ContextFlags = CONTEXT_CONTROL;

// Tell the system to get the registers associated with a thread.
GetThreadContext(hThread, &Context);

// The control register members in the CONTEXT structure
// reflect the thread's control registers. The other members
// are undefined.
```

注意，在调用GetThreadContext之前，首先必须对CONTEXT结构中的ContextFlags成员进行初始化。如果想要获得线程的控制寄存器和整数寄存器，应该像下面这样对ContextFlags进行初始化：

```
// Tell the system that we are interested
// in the control and integer registers.
Context.ContextFlags = CONTEXT_CONTROL | CONTEXT_INTEGER;
```

下面是一些标识符，使用这些标识符可以获得线程的所有重要的寄存器（即Microsoft视为最常用的那些寄存器）：

```
// Tell the system we are interested in the important registers.
Context.ContextFlags = CONTEXT_FULL;
```

在WinNT.h文件中，定义了CONTEXT\_FULL，请看表7-2。

表7-2 CONTEXT-FULL的意义

CPU类型	CONTEXT_FULL的定义
X86	CONTEXT_CONTROL   CONTEXT_INTEGER   CONTEXT_SEGMENTS
Alpha	CONTEXT_CONTROL   CONTEXT_FLOATING_POINT   CONTEXT_INTEGER

当GetThreadContext返回时，能够很容易地查看线程的任何寄存器的值，但是要记住，这意味着必须编写与CPU相关的代码。表7-3根据CPU类型列出了CONTEXT结构的指令指针和堆栈指针。

表7-3 CONTEXT结构的指令指针和堆栈指针

CPU类型	指令指针	堆栈指针
X86	CONTEXT.Eip	CONTEXT.Esp
Alpha	CONTEXT.Fir	CONTEXT.IntSp

Windows为编程人员提供了多么强大的功能啊！如果你认为它确实不错，那么你一定回喜

欢它的，因为Windows使你能够修改CONTEXT结构中的成员，然后通过调用SetThreadContext将新寄存器值放回线程的内核对象中：

```
BOOL SetThreadContext(
    HANDLE hThread,
    CONST CONTEXT *pContext);
```

同样，修改其环境的线程应该首先暂停，否则其结果将无法预测。

在调用SetThreadContext之前，必须再次对CONTEXT的ContextFlags成员进行初始化，如下面的代码所示：

```
CONTEXT Context;

// Stop the thread from running.
SuspendThread(hThread);

// Get the thread's context registers.
Context.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(hThread, &Context);
// Make the instruction pointer point to the address of your choice.
// Here I've arbitrarily set the address instruction pointer to
// 0x00010000.
#ifdef _ALPHA_
Context.Eip = 0x00010000;
#elif defined(_X86_)
Context.Eip = 0x00010000;
#else
#error Module contains CPU-specific code; modify and recompile.
#endif

// Set the thread's registers to reflect the changed values.
// It's not really necessary to reset the ControlFlags member
// because it was set earlier.
Context.ControlFlags = CONTEXT_CONTROL;
SetThreadContext(hThread, &Context);

// Resuming the thread will cause it to begin execution
// at address 0x00010000.
ResumeThread(hThread);
```

这有可能导致远程线程中的访问违规，向用户显示未处理的异常消息框，同时，远程进程终止运行。你将成功地终止另一个进程的运行，而你的进程则可以继续很好地运行。

GetThreadContext和SetThreadContext函数使你能够对线程进行许多方面的控制，但是在使用它们时应该小心。实际上，几乎没有应用程序调用这些函数。增加这些函数是为了增强调试程序和其他工具的功能。任何应用程序都可以调用它们。

第24章将详细地介绍CONTEXT结构。

## 7.7 线程的优先级

本章开头讲述了CPU是如何只使线程运行20ms，然后调度程序将另一个可调度的线程分配给CPU的。如果所有线程具有相同的优先级，那么就会发生这种情况，但是，在现实环境中，线程被赋予许多不同的优先级，这会影响到调度程序将哪个线程取出来作为下一个要运

行的线程。

每个线程都会被赋予一个从0（最低）到31（最高）的优先级号码。当系统确定将哪个线程分配给CPU时，它首先观察优先级为31的线程，并以循环方式对它们进行调度。如果优先级为31的线程可以调度，那么就将该线程赋予一个CPU。在该线程的时间片结束时，系统要查看是否还有另一个优先级为31的线程可以运行，如果有，它将允许该线程被赋予一个CPU。

只要优先级为31的线程是可调度的，系统就绝对不会将优先级为0到30的线程分配给CPU。这种情况称为渴求调度（starvation）。当高优先级线程使用大量的CPU时间，从而使得低优先级线程无法运行时，便会出现渴求情况。在多处理器计算机上出现渴求情况的可能性要少得多，因为在这样的计算机上，优先级为31和优先级为30的线程能够同时运行。系统总是设法使CPU保持繁忙状态，只有当没有线程可以调度的时候，CPU才处于空闲状态。

人们可能认为，在这样的系统中，低优先级线程永远得不到机会运行。不过正像前面指出的那样，在任何一个时段内，系统中的大多数线程是不能调度的。例如，如果进程的主线程调用GetMessage函数，而系统发现没有线程可以供它使用，那么系统就暂停进程的线程运行，释放该线程的剩余时间片，并且立即将CPU分配给另一个等待运行的线程。

如果没有为GetMessage函数显示可供检索的消息，那么进程的线程将保持暂停状态，并且决不会被分配给CPU。但是，当消息被置于线程的队列中时，系统就知道该线程不应该再处于暂停状态。此时，如果没有更高优先级的线程需要运行，系统就将该线程分配给一个CPU。

现在考虑另一个问题。高优先级线程将抢在低优先级线程之前运行，不管低优先级线程正在运行什么。例如，如果一个优先级为5的线程正在运行，系统发现一个高优先级的线程准备要运行，那么系统就会立即暂停低优先级线程的运行（即使它处于它的时间片中），并且将CPU分配给高优先级线程，使它获得一个完整的时间片。

还有，当系统引导时，它会创建一个特殊的线程，称为0页线程。该线程被赋予优先级0，它是整个系统中唯一的一个在优先级0上运行的线程。当系统中没有任何线程需要执行操作时，0页线程负责将系统中的所有空闲RAM页面置0。

## 7.8 对优先级的抽象说明

当Microsoft的开发人员设计线程调度程序时，他们发现该调度程序无法在所有时间适应所有人的需要。他们还发现，计算机的“作用”是不断变化的。当Windows NT问世时，对象链接和嵌入（OLE）应用程序还刚刚开始编写。现在，OLE应用程序已经司空见惯。游戏软件已经相当流行。当然，在Windows NT的早期，并没有更多地考虑Internet的问题。

调度算法对用户运行的应用程序类型有着相当大的影响。从一开始，Microsoft的开发人员就认识到，随着系统的用途的变化，他们必须不断修改调度算法。但是，软件开发人员需要在今天编写软件，而Microsoft则要保证软件能够在将来的系统版本上运行。那么Microsoft如何改变系统工作的方式并仍然保证软件能够运行呢？下面是解决这个问题的一些办法：

- Microsoft没有将调度程序的行为特性完全固定下来。
- Microsoft没有让应用程序充分利用调度程序的特性。
- Microsoft声称调度程序的算法是变化的，在编写代码时应有所准备。

Windows API展示了系统的调度程序上的一个抽象层，这样就永远不会直接与调度程序进行通信。相反，要调用Windows函数，以便根据运行的系统版本“转换”参数。本章将介绍这个抽象层。

当设计一个应用程序时，你应该考虑到还有什么别的应用程序会与你的应用程序一道运行。

然后，应该根据你的应用程序中的线程应该具备何种响应性，选择一个优先级类。这听起来有些费解，不过情况确实如此。Microsoft不想作出任何将来可能影响你的代码运行的承诺。

Windows支持6个优先级类：即空闲、低于正常、正常、高于正常、高和实时。当然，正常优先级是最常用的优先级类，99%的应用程序均使用这个优先级类。表7-4描述了这些优先级类。

表7-4 Windows 支持的优先级类

优先级类	描述
实时	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。该进程中的线程还会抢先于操作系统组件之前运行。使用本优先级类时必须极端小心
高	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。Task Manager（任务管理器）在这个类上运行，以便用户可以撤消脱离控制的进程
高于正常	进程中的线程在正常优先级与高优先级之间运行（这是 Windows 2000中的新优先级类）
正常	进程中的线程没有特殊的调度需求
低于正常	进程中的线程在正常优先级与空闲优先级之间运行（这是 Windows 2000中的新优先级类）
空闲	进程中的线程在系统空闲时运行。该进程通常由屏幕保护程序或后台实用程序和搜集统计数据的软件使用

当系统什么也不做的时候，将空闲优先级类用于应用程序的运行是最恰当不过的。没有用交互方式使用的计算机有可能仍然很繁忙（比如作为文件服务器），不应该与屏幕保护程序争用CPU时间。定期更新系统的某些状态的统计信息跟踪应用程序不应该干扰关键任务的运行。

只有当绝对必要的时候，才可以使用高优先级类。你会惊奇地发现，Windows Explorer是在高优先级上运行的。大多数时间 Explorer的线程是暂停的，等待用户按下操作键或者点击鼠标按钮时被唤醒。当 Explorer的线程处于暂停状态时，系统不将它的线程分配给 CPU。因为这将使低优先级线程得以运行。但是一旦用户按下一个操作键或组合键，如 Ctrl+Esc，系统就会唤醒 Explorer的线程（当用户按下 Ctrl+Esc组合键时，也会出现 Start菜单）。如果低优先级线程正在运行，系统会立即抢在这些线程的前面，让 Explorer的线程优先运行。

Microsoft就是按这种方法设计 Explorer的，因为用户希望无论系统中正在运行什么，外壳程序都具有极强的响应能力。实际上，即使低优先级线程在无限循环中暂停运行，也能显示 Explorer的窗口。由于 Explorer的线程拥有较高的优先级，因此执行无限循环的线程被抢占，Explorer让用户终止挂起进程的运行。Explorer的运行特性非常出色，大部分时间它的线程无事可做，不必占用CPU时间。如果情况不是如此，那么整个系统的运行速度就会慢得多，许多应用程序就不会作出响应。

应该尽可能避免使用实时优先级类。实际上 Windows NT 3.1的早期测试版并没有向应用程序展示这个优先级类，尽管该操作系统支持这个类。实时优先级是很高的优先级，它可能干扰操作系统任务的运行，因为大多数操作系统线程均以较低的优先级来运行。因此实时线程可能阻止必要的磁盘 I/O信息和网络信息的产生。此外，键盘和鼠标输入将无法及时得到处理，用户可能以为系统已经暂停运行。大体来说，必须有足够的理由才能使用实时优先级，比如需要以很短的等待时间来响应硬件事件，或者执行某些不能中断的短期任务。

注意 除非用户拥有“提高调度优先级”的权限，否则进程不能用实时优先级类来运



行。凡是被指定为管理员或特权用户的用户，均默认拥有该权限。

当然，大多数进程都属于正常优先级类。低于正常和高于正常的优先级类是 Windows 2000 中的新增优先级。Microsoft 增加这些优先级类的原因是，有若干家公司抱怨现有的优先级类无法提供足够的灵活性。

一旦选定了优先级类之后，就不必考虑你的应用程序与其他应用程序之间的关系，只需要集中考虑你的应用程序中的各个线程。Windows 支持 7 个相对的线程优先级：即空闲、最低、低于正常、正常、高于正常、最高和关键时间优先级。这些优先级是相对于进程的优先级类而言的。大多数线程都使用正常线程优先级。表 7-5 描述了这些相对的线程优先级。

表 7-5 相对的线程优先级

相对的线程优先级	描 述
关键时间	对于实时优先级类来说，线程在优先级 31 上运行，对于其他优先级类来说，线程在优先级 15 上运行
最高	线程在高于正常优先级的上两级上运行
高于正常	线程在正常优先级的上一级上运行
正常	线程在进程的优先级类上正常运行
低于正常	线程在低于正常优先级的下一级上运行
最低	线程在低于正常优先级的下两级上运行
空闲	对于实时优先级类来说，线程在优先级 16 上运行对于其他优先级类来说，线程在优先级 1 上运行

概括起来说，进程是优先级类的一个组成部分，你为进程中的线程赋予相对线程优先级。这里没有讲到 0 到 31 的优先级的任何情况。应用程序开发人员从来不必具体设置优先级。相反，系统负责将进程的优先级类和线程的相对优先级映射到一个优先级上。正是这种映射方式，Microsoft 不想拘泥不变。实际上这种映射方式是随着系统的版本的升级而变化的。

表 7-6 显示了这种映射方式是如何用于 Windows 2000 的，注意，Windows NT 的早期版本和某些 Windows 95 和 Windows 98 版本采用了不同的映射方式。未来的 Windows 版本中的映射方式也会变化。

例如，正常进程中的正常线程被赋予的优先级是 8。由于大多数进程属于正常优先级类，而大多数线程属于正常线程优先级，因此系统中的大多数线程的优先级是 8。

如果高优先级进程中有一个正常线程，该线程的优先级将是 13。如果将进程的优先级类改为 8，那么线程的优先级就变为 4。如果改变了进程的优先级类，线程的相对优先级不变，但是它的优先级的等级却发生了变化。

表 7-6 进程优先级类和线程相对优先级的映射

相 对 线 程 优 先 级	空闲	低于 正常	正常	高于 正常	高	实时
关键时间	15	15	5	15	15	31
最高	6	8	10	12	15	26
高于正常	5	7	9	11	14	25
正常	4	6	8	10	13	24
低于正常	3	5	7	9	12	23
最低	2	4	6	8	11	22
空闲	1	1	1	1	1	16



注意，表7-6并没有显示优先级的等级为0的线程。这是因为0优先级保留供零页线程使用，系统不允许任何其他线程拥有0优先级。另外，下列优先级等级是无法使用的：17、18、19、20、21、27、28、29和30。如果编写一个以内核方式运行的设备驱动程序，可以获得这些优先级等级，而用户方式的应用程序则不能。另外还要注意，实时优先级类中的线程不能低于优先级等级16。同样，非实时优先级类中的线程的等级不能高于15。

注意 有些人常常搞不清进程优先级类的概念。他们认为这可能意味着进程是可以调度的。但是进程是根本不能调度的，只有线程才能被调度。进程优先级类是个抽象概念，Microsoft提出这个概念的目的，是为了帮助你将它与调度程序的内部运行情况区分开来。它没有其他目的。

注意 一般来说，大多数时候高优先级的线程不应该处于可调度状态。当线程要进行某种操作时，它能迅速获得CPU时间。这时线程应该尽可能少地执行CPU指令，并返回睡眠状态，等待再次变成可调度状态。相反，低优先级的线程可以保持可调度状态，执行大量的CPU指令来进行它的操作。如果按照这些原则来办，整个操作系统就能正确地对用户作出响应。

## 7.9 程序的优先级

进程是如何被赋予优先级类的呢？当调用CreateProcess时，可以在fdwCreate参数中传递需要的优先级类。表7-7显示了优先级类的标识符。

表7-7 优先级类的标识类

优先级类	标识符
实时	REALTIME_PRIORITY_CLASS
高	HIGH_PRIORITY_CLASS
高于正常	ABOVE_NORMAL_PRIORITY_CLASS
正常	NORMAL_PRIORITY_CLASS
低于正常	BELOW_NORMAL_PRIORITY_CLASS
空闲	IDLE_PRIORITY_CLASS

创建子进程的进程负责选择子进程运行的优先级类，这看起来有点奇怪。让我们以Explorer为例来说明这个问题。当使用Explorer来运行一个应用程序时，新进程按正常优先级运行。Explorer不知道进程在做什么，也不知道隔多长时间它的线程需要进行调度。但是，一旦子进程运行，它能够通过调用SetPriorityClass来改变它自己的优先级类：

```
BOOL SetPriorityClass(  
    HANDLE hProcess,  
    DWORD fdwPriority);
```

该函数将hProcess标识的优先级类改为fdwPriority参数中设定的值。fdwPriority参数可以是表7-7显示的标识符之一。由于该函数带有一个进程句柄，因此，只要拥有该进程的句柄和足够的访问权，就能够改变系统中运行的任何进程的优先级类。

一般来说，进程将试图改变它自己的优先级类。下面是如何使一个进程将它自己的优先级类设置为空闲的例子：

```
BOOL SetPriorityClass(  
    GetCurrentProcess(),  
    IDLE_PRIORITY_CLASS);
```

下面是用来检索进程的优先级类的补充函数：

```
DWORD GetPriorityClass(HANDLE hProcess);
```

正如你所期望的那样，该函数将返回表 7-7 中列出的标识符之一。

当使用命令外壳启动一个程序时，该程序的起始优先级是正常优先级。但是，如果使用 Start 命令来启动该程序，可以使用一个开关来设定应用程序的起始优先级。例如，在命令外壳输入下面的命令可使系统启动 Calculator，并在开始时按空闲优先级来运行它：

```
C:\>START /LOW CALC.EXE
```

Start 命令还能识别 /BELOWNORMAL、/NORMAL、/ABOVENORMAL、/HIGH 和 /REALTIME 等开关，以便按它们各自的优先级启动执行一个应用程序。当然，一旦应用程序启动运行，它就可以调用 SetPriorityClass 函数，将它自己的优先级改为它选择的任何优先级。

**Windows 98** Windows 98 的 Start 命令并不支持这些开关中的任何一个。Windows 98 命令外壳启动的进程总是使用正常优先级类来运行。

Windows 2000 的 Task Manager 使得用户可以改变进程的优先级类。图 7-2 显示了 Task Manager 的 Processes 选项卡，它显示了当前运行的所有进程。Base Pri 列显示了每个进程的优先级类。可以改变进程的优先级类，方法是选定一个进程，然后从上下文菜单的 Set Priority（设置优先级）子菜单中选择一个选项。

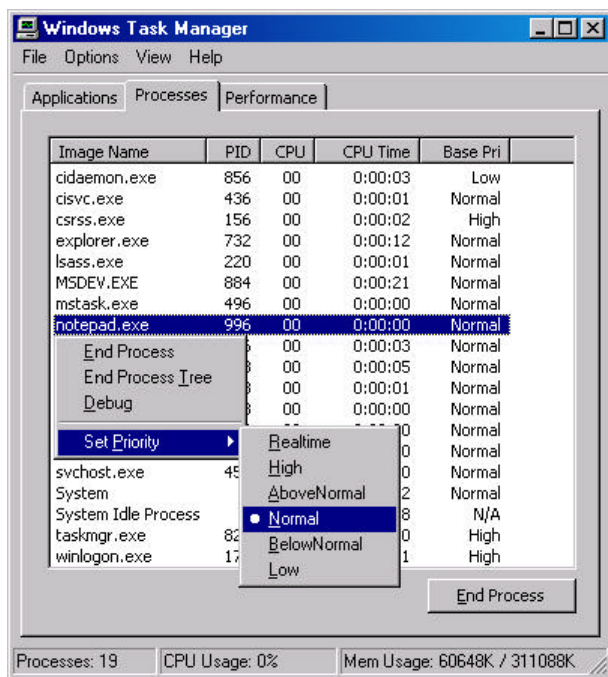


图7-2 Windows Task Manager 对话框

当一个线程刚刚创建时，它的相对线程优先级总是设置为正常优先级。我总感到有些奇怪，CreateThread 没有为调用者提供一个设置新线程的相对优先级的方法。若要设置和获得线程的相对优先级，必须调用下面的这些函数：

```
BOOL SetThreadPriority(  
    HANDLE hThread,  
    int nPriority);
```

当然，hThread参数用于标识想要改变优先级的单个线程，nPriority参数是表7-8列出的7个标识符之一。

表7-8 线程相对优先级的标识符常量

相对线程优先级	标识符常量
关键时间	THREAD_PRIORITY_TIME_CRITICAL
最高	THREAD_PRIORITY_HIGHEST
高于正常	THREAD_PRIORITY_ABOVE_NORMAL
正常	THREAD_PRIORITY_NORMAL
低于正常	THREAD_PRIORITY_BELOW_NORMAL
最低	THREAD_PRIORITY_LOWEST
空闲	THREAD_PRIORITY_IDLE

下面是检索线程的相对优先级的补充函数：

```
int GetThreadPriority(HANDLE hThread);
```

该函数返回表7-8列出的标识符之一。

若要创建一个带有相对优先级为空闲的线程，可以执行类似下面的代码：

```
DWORD dwThreadId;  
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,  
    CREATE_SUSPENDED, &dwThreadId);  
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);  
ResumeThread(hThread);  
CloseHandle(hThread);
```

注意，CreateThread函数创建的新函数带有的相对优先级总是正常优先级。若要使线程以空闲优先级来运行，应该将CREATE\_SUSPENDED标志传递给CreateThread函数，这可以防止线程执行任何代码。然后可以调用 SetThreadPriority，将线程的优先级改为相对空闲优先级。这时可以调用 ResumeThread，使得线程成为可调度的线程。你不知道线程何时能够获得 CPU 时间，但是调度程序会考虑这样一个情况，即该线程拥有一个空闲优先级。最后，可以关闭新线程的句柄，一旦线程终止运行，内核对象就能被撤消。

注意 Windows 没有提供返回线程的优先级的函数。这是故意进行的。记住，Microsoft 保留了随时修改调度算法的权利。你不会设计需要调度算法专门知识的应用程序。如果坚持使用进程优先级类和相对线程优先级，你的应用程序不仅现在能够顺利地运行，而且在系统的将来版本上也能很好地运行。

### 7.9.1 动态提高线程的优先级等级

通过将线程的相对优先级与线程的进程优先级类综合起来考虑，系统就可以确定线程的优先级等级。有时这称为线程的基本优先级等级。系统常常要提高线程的优先级等级，以便对窗口消息或读取磁盘等 I/O 事件作出响应。

例如，在高优先级类进程中的一个正常优先级等级的线程的基本优先级等级是 13。如果用户按下一个操作键，系统就会将一个 WM\_KEYDOWN 消息放入线程的队列中。由于一个消息已经出现在线程的队列中，因此该线程就是可调度的线程。此外，键盘设备驱动程序也能够告诉系统暂时提高线程的优先级等级。该线程的优先级等级可能提高 2 级，其当前优先级等级改为 15。

系统在优先级为15时为一个时间片对该线程进行调度。一旦该时间片结束，系统便将线程的优先级递减1，使下一个时间片的线程优先级降为14。该线程的第三个时间片按优先级等级13来执行。如果线程要求执行更多的时间片，均按它的基本优先级等级13来执行。

注意，线程的当前优先级等级决不会低于线程的基本优先级等级。此外，导致线程成为可调度线程的设备驱动程序可以决定优先级等级提高的数量。Microsoft并没有规定各个设备驱动程序可以给线程的优先级提高多少个等级。这样就使得Microsoft可以不断地调整线程优先级提高的动态等级，以确定最佳的总体响应性能。

系统只能为基本优先级等级在1至15之间的线程提高其优先级等级。实际上这是因为这个范围称为动态优先级范围。此外，系统决不会将线程的优先级等级提高到实时范围（高于15）。由于实时范围中的线程能够执行大多数操作系统的函数，因此给等级的提高规定一个范围，就可以防止应用程序干扰操作系统的运行。另外，系统决不会动态提高实时范围内的线程优先级等级。

有些编程人员抱怨说，系统动态提高线程优先级等级的功能对他们的线程性能会产生一种不良的影响，为此Microsoft增加了下面两个函数，这样就能够使系统的动态提高线程优先级等级的功能不起作用：

```
BOOL SetProcessPriorityBoost(  
    HANDLE hProcess,  
    BOOL DisablePriorityBoost);  
BOOL SetThreadPriorityBoost(  
    HANDLE hThread,  
    BOOL DisablePriorityBoost);
```

SetProcessPriorityBoost负责告诉系统激活或停用进行中的所有线程的优先级提高功能，而SetThreadPriorityBoost则让你激活或停用各个线程的优先级提高功能。这两个函数具有许多相似的共性，可以用来确定是激活还是停用优先级提高功能：

```
BOOL GetProcessPriorityBoost(  
    HANDLE hProcess,  
    PBOOL pDisablePriorityBoost);  
BOOL GetThreadPriorityBoost(  
    HANDLE hThread,  
    PBOOL pDisablePriorityBoost);
```

对于这两个函数中的每个函数，可以传递想要查询的进程或线程的句柄，以及由函数设置的BOOL的地址。

Windows 98 Windows 98没有提供这4个函数的有用的实现代码。它们全部返回FALSE，后来对GetLastError的调用将返回ERROR\_CALL\_NOT\_IMPLEMENTED。

另一种情况也会导致系统动态地提高线程的优先级等级。比如有一个优先级为4的线程准备运行但是却不能运行，因为一个优先级为8的线程正连续被调度。在这种情况下，优先级为4的线程就非常渴望得到CPU时间。当系统发现一个线程在大约3至4s内一直渴望得到CPU时间，它就将这个渴望得到CPU时间的线程的优先级动态提高到15，并让该线程运行两倍于它的时间量。当到了两倍时间量的时候，该线程的优先级立即返回到它的基本优先级。

## 7.9.2 为前台进程调整调度程序

当用户对进程的窗口进行操作时，该进程就称为前台进程，所有其他进程则称为后台进程。当然，用户希望他正在使用的进程比后台进程具有更强的响应性。为了提高前台进程的响应性，

Windows能够为前台进程中的线程调整其调度算法。对于 Windows 2000来说,系统可以为前台进程的线程提供比通常多的CPU时间量。这种调整只能在前台进程属于正常优先级类的进程时才能进行。如果它属于其他任何优先级类,就无法进行任何调整。

Windows 2000实际上允许用户对这种调整进行相应的配置。在 System Properties (系统属性)对话框的 Advanced选项卡上,用户可以单击 Performance Options(性能选项)按钮,打开图 7-3所示的对话框。

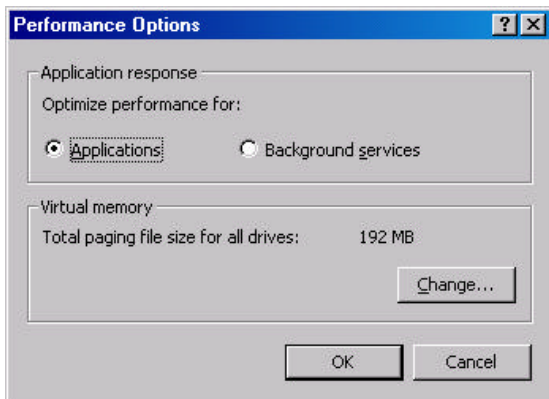


图7-3 Performance Options 对话框

如果用户选择优化应用程序的性能,系统就执行配置的调整。如果用户选择优化后台服务程序的性能,系统就不进行调整。当安装 Windows 2000的专业版时, Applications就会被默认选定。对于 Windows 2000的所有其他版本,则默认选定 Background Services,因为计算机将主要由非交互式用户使用。

当进程移到前台时, Windows 98也会对正常优先级类的进程中的线程调度算法进行调整。当一个优先级为正常的进程移到前台时,系统便将最低、低于正常、正常、高于正常和最高等优先级的线程的优先级提高 1,优先级为空闲和关键时间的线程的优先级则不予提高。因此,在正常优先级类的进程中运行的、其相对优先级为正常的线程,它的优先级等级是 9而不是8。当进程返回后台时,进程中的线程便自动返回它们定义好的基本优先级等级。

**Windows 98** Windows 98没有提供允许用户配置这种调整手段的任何用户界面,因为 Windows 98不是作为专用服务器来运行的。

将进程改为前台进程的原因是,使它们能够对用户的输入更快地作出响应。如果不改为前台进程,那么在后台的正常打印进程与在后台接收用户输入的正常进程就会平等地争用 CPU时间。用户会发现文本无法在前台应用程序中顺利地显示。但是,由于系统改变了前台进程的线程优先级,前台进程的线程就能对用户的输入更好地作出响应。

### 7.9.3 Scheduling Lab示例应用程序

使用Scheduling Lab应用程序“07SchedLab.exe”(见后面的清单 7-1),可以对进程优先级类和相对线程优先级进行操作试验,以了解它们对系统的总体性能产生的影响。该应用程序的源代码和源文件位于本书所附光盘上的 07-SchedLab目录中。当启动该程序时,就会出现图 7-4所示的窗口。

开始时,主线程总是处于繁忙状态,因此 CPU的使用量立即跳到 100%。该主线程连续递增一个数字,并将它添加到右边的列表框。这个数字并没有任何意义,它只是显示线程正在忙



于进行什么操作。若要了解线程调度对系统会产生什么实际影响,建议至少要同时运行该示例应用程序的两个实例,看一看改变一个实例的优先级会对另一个实例带来的影响。也可以运行 Task Manager,以便监控所有实例的CPU使用量。

当进行这些测试时,CPU的使用量开始时将上升为100%,该应用程序的所有实例将获得大约相等的CPU时间(Task Manager应该显示应用程序的所有实例大致相同的CPU使用量百分比)。

如果将一个实例的优先级类改为高于正常或高优先级类,那么应该看到它得到了大部分的CPU使用量。而其他实例中的数字滚动则没有规律。但是其他实例的数字不会完全停止滚动,因为系统将为渴求CPU时间的线程自动执行优先级的动态提高。不管怎样,可以随意调整优先级类和相对线程优先级,以了解它们对其他实例的影响。我有目的地对 Scheduling Lab应用程序进行了编码,这样就无法将进程改为实时优先级类,这可以防止操作系统线程的不正常的运行。如果想要试用实时优先级,必须自己修改源代码。

可以使用Sleep域,使主线程在0到9999之间的任意毫秒内无法调度。请试用这项功能,并观察传递仅为1ms的睡眠值时可以重新获得多少CPU时间。在我的300MHz Pentium II笔记本电脑上,我赢得了99%的CPU时间。

单击Suspend(暂停)按钮,可使主线程产生一个子线程。这个子线程能够暂停主线程的运行,并显示图7-5所示的消息框。

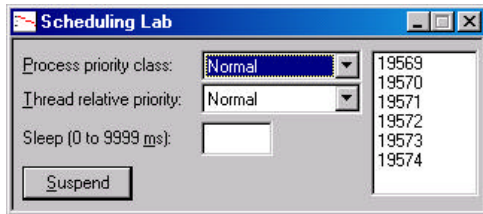


图7-4 进程优先级类和相对线程优先级的试验窗口



图7-5 消息框

当这个消息框显示时,主线程将完全暂停运行,并且不使用任何CPU时间。子线程也不使用任何CPU时间,因为它是在等待用户执行某种操作。当消息框显示时,可以将它移到应用程序的主窗口,然后将它移开,这样就能够看到主窗口。由于主线程已经暂停运行,因此主窗口将无法接收任何窗口消息(包括WM\_PAINT)。这证明该线程已经暂停运行。当关闭该消息框时,主线程就恢复运行,CPU使用量回到100%。

若要再进行一次试验,请打开前一节介绍的Performance Options对话框,将Application改为Background Services,或者将Background Services改为Application。然后打开SchedLab程序的多个实例,将它们全部设置为正常优先级类,并激活其中的一个,使之成为一个前台进程。这时就能够看到性能的设置对前台/后台进程产生的影响。

清单7-1 SchedLab示例应用程序



## SchedLab.cpp

```
/*  
*****  
Module: SchedLab.cpp
```



Notices: Copyright (c) 2000 Jeffrey Richter

\*\*\*\*\*/

```
#include "..\CmnHdr.H"      /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>        // For _beginthreadex
#include "Resource.H"
```

//

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    HANDLE hThreadPrimary = (HANDLE) pvParam;
    SuspendThread(hThreadPrimary);
    chMB(
        "The Primary thread is suspended.\n"
        "It no longer responds to input and produces no output.\n"
        "Press OK to resume the primary thread & exit this secondary thread.\n");
    ResumeThread(hThreadPrimary);
    CloseHandle(hThreadPrimary);

    // To avoid deadlock, call EnableWindow after ResumeThread.
    EnableWindow(
        GetDlgItem(FindWindow(NULL, TEXT("Scheduling Lab")), IDC_SUSPEND),
        TRUE);
    return(0);
}
```

//

```
BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_SCHEDLAB);
    // Initialize process priority classes
    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSPRIORITYCLASS);

    int n = ComboBox_AddString(hwndCtl, TEXT("High"));
    ComboBox_SetItemData(hwndCtl, n, HIGH_PRIORITY_CLASS);

    // Save our current priority class
    DWORD dwpc = GetPriorityClass(GetCurrentProcess());

    if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

        // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

        // Restore our original priority class
        SetPriorityClass(GetCurrentProcess(), dwpc);

        // Add the Above Normal priority class
```

```

        n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
        ComboBox_SetItemData(hwndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);

        dwpc = 0; // Remember that this system supports below normal
    }

    int nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
    ComboBox_SetItemData(hwndCtl, n, NORMAL_PRIORITY_CLASS);

    if (dwpc == 0) {

        // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

        // Add the Below Normal priority class
        n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
        ComboBox_SetItemData(hwndCtl, n, BELOW_NORMAL_PRIORITY_CLASS);
    }

    n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
    ComboBox_SetItemData(hwndCtl, n, IDLE_PRIORITY_CLASS);

    ComboBox_SetCurSel(hwndCtl, nNormal);

    // Initialize thread relative priorities
    hwndCtl = GetDlgItem(hwnd, IDC_THREADRELATIVEPRIORITY);

    n = ComboBox_AddString(hwndCtl, TEXT("Time critical"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_TIME_CRITICAL);
    n = ComboBox_AddString(hwndCtl, TEXT("Highest"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_HIGHEST);

    n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_ABOVE_NORMAL);

    nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_NORMAL);

    n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_BELOW_NORMAL);

    n = ComboBox_AddString(hwndCtl, TEXT("Lowest"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_LOWEST);

    n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
    ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_IDLE);

    ComboBox_SetCurSel(hwndCtl, nNormal);

    Edit_LimitText(GetDlgItem(hwnd, IDC_SLEEPTIME), 4); // Maximum of 9999

    return(TRUE);
}

```

```

////////////////////////////////////

```

```

voidDlg_OnCommand (HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            PostQuitMessage(0);
            break;

        case IDC_PROCESSPRIORITYCLASS:
            if (codeNotify == CBN_SELCHANGE) {
                SetPriorityClass(GetCurrentProcess(), (DWORD)
                    ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
            }
            break;
        case IDC_THREADRELATIVEPRIORITY:
            if (codeNotify == CBN_SELCHANGE) {
                SetThreadPriority(GetCurrentThread(), (DWORD)
                    ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
            }
            break;

        case IDC_SUSPEND:
            // To avoid deadlock, call EnableWindow before creating
            // the thread which calls SuspendThread.
            EnableWindow(hwndCtl, FALSE);

            HANDLE hThreadPrimary;
            DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
                GetCurrentProcess(), &hThreadPrimary,
                THREAD_SUSPEND_RESUME, FALSE, DUPLICATE_SAME_ACCESS);
            DWORD dwThreadId;
            CloseHandle(chBEGINTHREADEX(NULL, 0, ThreadFunc,
                hThreadPrimary, 0, &dwThreadId));
            break;
    }
}

/////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }

    return(FALSE);
}

/////////////////////////////////////////////////////////////////

int WINAPI _tWinMain (HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

```

```

HWND hwnd =
    CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_SCHEDLAB), NULL, Dlg_Proc);
BOOL fQuit = FALSE;

while (!fQuit) {
    MSG msg;
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

        // IsDialogMessage allows keyboard navigation to work properly.
        if (!IsDialogMessage(hwnd, &msg)) {

            if (msg.message == WM_QUIT) {
                fQuit = TRUE; // For WM_QUIT, terminate the loop.
            } else {
                // Not a WM_QUIT message. Translate it and dispatch it.
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        } // if (!IsDialogMessage())
    } else {

        // Add a number to the listbox
        static int s_n = -1;
        TCHAR sz[20];
        wsprintf(sz, TEXT("%u"), ++s_n);
        HWND hwndWork = GetDlgItem(hwnd, IDC_WORK);
        ListBox_SetCurSel(hwndWork, ListBox_AddString(hwndWork, sz));

        // Remove some strings if there are too many entries
        while (ListBox_GetCount(hwndWork) > 100)
            ListBox_DeleteString(hwndWork, 0);

        // How long should the thread sleep
        int nSleep = GetDlgItemInt(hwnd, IDC_SLEEPTIME, NULL, FALSE);
        if (chINRANGE(1, nSleep, 9999))
            Sleep(nSleep);
    }
}
DestroyWindow(hwnd);
return(0);
}

```

//////////////////////////////////// End of File //////////////////////////////////////

## SchedLab.rc

```

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//

```

```

#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#ifdef !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////////////////////
//
// Dialog
//

IDD_SCHEDLAB_DIALOGEX 0, 0, 209, 70
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
EXSTYLE WS_EX_NOPARENTNOTIFY | WS_EX_CLIENTEDGE
CAPTION "Scheduling Lab"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT                "&Process priority class:", IDC_STATIC, 4, 6, 68, 8
    COMBOBOX              IDC_PROCESSPRIORITYCLASS, 84, 4, 72, 80, CBS_DROPDOWNLIST |
        WS_TABSTOP
    LTEXT                "&Thread relative priority:", IDC_STATIC, 4, 20, 72, 8
    COMBOBOX              IDC_THREADRELATIVEPRIORITY, 84, 18, 72, 76, CBS_DROPDOWNLIST |
        WS_TABSTOP
    LTEXT                "Sleep (0 to 9999 &ms):", IDC_STATIC, 4, 36, 68, 8
    EDITTEXT              IDC_SLEEP, 84, 34, 32, 14, ES_NUMBER
    PUSHBUTTON            "&Suspend", IDC_SUSPEND, 4, 52, 49, 14
    LISTBOX                IDC_WORK, 160, 4, 48, 60, NOT LBS_NOTIFY |
        LBS_NOINTEGRALHEIGHT | LBS_NOSEL | WS_TABSTOP
END

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_SCHEDLAB, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 202
        TOPMARGIN, 7
        BOTTOMMARGIN, 63
    END
END

```

```

END
#endif // APSTUDIO_INVOKED

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif // APSTUDIO_INVOKED

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SCHEDLAB          ICON    DISCARDABLE    "SchedLab.ico"
#endif // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

## 7.10 亲缘性

按照默认设置，当系统将线程分配给处理器时，Windows 2000使用软亲缘性来进行操作。这意味着如果所有其他因素相同的话，它将设法在它上次运行的那个处理器上运行线程。让线程留在单个处理器上，有助于重复使用仍然在处理器的内存高速缓存中的数据。



有一种新的计算机结构，称为 NUMA（非统一内存访问），在该结构中，计算机包含若干块插件板，每个插件板上有 4 个 CPU 和它自己的内存区。图 7-6 显示了一台配有 3 块插件板的计算机，总共有 12 个 CPU，这样，任何一个线程都可以在 12 个 CPU 中的任何一个上运行。

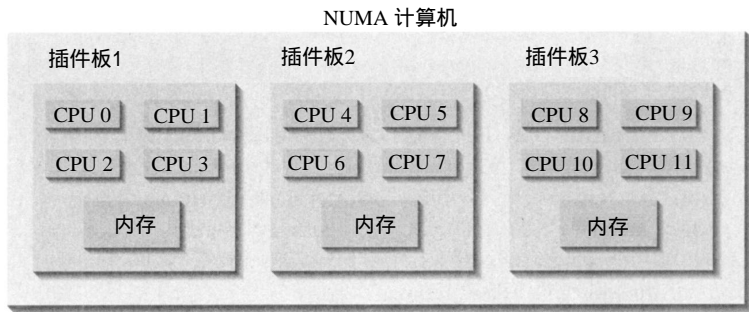


图7-6 NUMA 计算机结构示意图

当 CPU 访问的内存是它自己的插件板上的内存时，NUMA 系统运行的性能最好。如果 CPU 需要访问位于另一个插件板上的内存时，性能就会大大降低。在这样的环境中，就需要来自一个进程中的线程在 CPU 0 至 3 上运行，让另一个进程中的线程在 CPU 4 至 7 上运行，依次类推。为了适应这种计算机结构的需要，Windows 2000 允许设置进程和线程的亲缘性。换句话说，可以控制哪个 CPU 能够运行某些线程。这称为硬亲缘性。

计算机在引导时，系统要确定机器中有多少个 CPU 可供使用。通过调用 `GetSystemInfo` 函数（第 14 章介绍），应用程序就能查询机器中的 CPU 数量。按照默认设置，任何线程都可以调度到这些 CPU 中的任何一个上去运行。为了限制在可用 CPU 的子集上运行的单个进程中的线程数量，可以调用 `SetProcessAffinityMask`：

```
BOOL SetProcessAffinityMask(  
    HANDLE hProcess,  
    DWORD_PTR dwProcessAffinityMask);
```

第一个参数 `hProcess` 用于指明要影响的是哪个进程。第二个参数 `dwProcessAffinityMask` 是个位屏蔽，用于指明线程可以在哪些 CPU 上运行。例如，传递 `0x00000005` 表示该进程中的线程可以在 CPU 0 和 CPU 2 上运行，但是不能在 CPU 1 和 CPU 3 至 31 上运行。

注意，子进程可以继承进程的亲缘性。因此，如果一个进程的亲缘性屏蔽是 `0x00000005`，那么它的子进程中的任何线程都拥有相同的位屏蔽，并共享相同的 CPU。此外，可以使用作业内核对象将一组进程限制在要求的一组 CPU 上运行。

当然，还有一个函数也能够返回进程的亲缘性位屏蔽，它就是 `GetProcessAffinityMask`，如下面的代码所示：

```
BOOL GetProcessAffinityMask(  
    HANDLE hProcess,  
    PDWORD_PTR pdwProcessAffinityMask,  
    PDWORD_PTR pdwSystemAffinityMask);
```

这里也可以传递想要亲缘性屏蔽的进程句柄，该函数填入 `pdwProcessAffinityMask` 指向的变量。该函数还能返回系统的亲缘性屏蔽（在 `pdwSystemAffinityMask` 指向的变量中）。系统的亲缘性屏蔽用于指明系统的哪个 CPU 能够处理线程。进程的亲缘性屏蔽始终是一个系统的亲缘性屏蔽的正确子集。

Windows 98 无论计算机中实际拥有多少个CPU，Windows 98只使用一个CPU。因此，GetProcessAffinityMask总是用1填入两个变量中。

到现在为止，已经介绍了如何将进程的多个线程限制到一组CPU上去运行。有时可能想要将进程中的一个线程限制到一组CPU上去运行。例如，可能有一个包含4个线程的进程，它们在拥有4个CPU的计算机上运行。如果这些线程中的一个线程正在执行非常重要的操作，而你想增加某个CPU始终可供它使用的可能性，为此你对其他3个线程进行了限制，使它们不能在CPU 0上运行，而只能在CPU 1、2和3上运行。

通过调用SetThreadAffinityMask，就能为各个线程设置亲缘性屏蔽：

```
DWORD_PTR SetThreadAffinityMask(  
    HANDLE hThread,  
    DWORD_PTR dwThreadAffinityMask);
```

该函数中的hThread参数用于指明要限制哪个线程，dwThreadAffinityMask用于指明该线程能够在哪个CPU上运行。dwThreadAffinityMask必须是进程的亲缘性屏蔽的相应子集。返回值是线程的前一个亲缘性屏蔽。因此，若要将3个线程限制到CPU 1、2和3上去运行，可以这样操作：

```
// Thread 0 can only run on CPU 0.  
SetThreadAffinityMask(hThread0, 0x00000001);  
  
// Threads 1, 2, 3 run on CPUs 1, 2, 3.  
SetThreadAffinityMask(hThread1, 0x0000000E);  
SetThreadAffinityMask(hThread2, 0x0000000E);  
SetThreadAffinityMask(hThread3, 0x0000000E);
```

Windows 98 由于计算机中无论配有多少个CPU，Windows 98只使用一个CPU，因此dwThreadAffinityMask参数必须始终是1。

当一个x86系统引导时，系统要执行相应的代码，以便测定主机上的哪些CPU遇到了著名的Pentium浮点错误。系统必须为每个CPU测试其浮点错误，方法是将线程的亲缘性设置为第一个CPU，执行潜在的故障分割操作，并将结果与已知的正确答案进行比较。然后对下一个CPU进行上述同样的操作，如此等等。

注意 在大多数环境中，改变线程的亲缘性就会影响调度程序有效地在各个CPU之间移植线程的能力，而这种能力可以最有效地使用CPU时间。表7-9显示了一个例子。

表7-9 线程的亲缘性示例

线 程	优 先 级	亲缘性屏蔽	结 果
A	4	0x00000001	CPU 0
B	8	0x00000003	CPU 1
C	6	0x00000002	不能运行

当线程A被唤醒时，调度程序发现该线程可以在CPU 0上运行，因此它被分配给CPU 0。然后线程B被唤醒，调度程序发现该线程可以被分配给CPU 0或1，但是，由于CPU 0正在使用之中，因此调度程序将线程B分配给了CPU 1。至此，一切进行得很顺利。

这时线程C被唤醒，调度程序发现它只能在CPU 1上运行。但是CPU 1正在被线程

B使用着，它是个优先级为8的线程。由于线程C的优先级为6，因此它不能抢在线程B的前面运行。线程C可以抢在线程A的前面运行，因为线程A的优先级是4，但是调度程序不会使它抢在线程A的前面运行，因为线程C不能在CPU 0上运行。

这显示出为线程设置硬亲缘性将会对调度程序的优先级设置方案产生什么样的影响。

有时强制将一个线程分配给特定的CPU的做法是不妥当的。例如，有3个线程都只能在CPU 0上运行，而CPU 1、2和3则闲着无事可做。在这种情况下，如果告诉系统想让一个线程在某个CPU上运行，但是允许该线程在可能的情况下移到另一个CPU上去运行，那么这种办法会更好些。

若要为线程设置一个理想的CPU，可以调用SetThreadIdealProcessor:

```
DWORD SetThreadIdealProcessor(
    HANDLE hThread,
    DWORD dwIdealProcessor);
```

hThread用于指明要为哪个线程设置首选的CPU。与我们已经介绍的其他函数不同，dwIdealProcessor函数不是个位屏蔽函数，它是个从0到31的整数，用于指明供线程使用的首选CPU。可以传递一个MAXIMUM\_PROCESSORS的值（在WinNT.h中定义为32），用于指明不存在理想的CPU。如果没有为该线程设置理想的CPU，那么该函数返回前一个理想的CPU或MAXIMUM\_PROCESSORS。

也可以在一个可执行文件的头上设置处理器亲缘性。奇怪的是，似乎不存在它的链接程序开关，不过可以使用类似下面的代码：

```
// Load the EXE into memory.
PLOADED_IMAGE pLoadedImage = ImageLoad(szExeName, NULL);

// Get the current load configuration information for the EXE.
IMAGE_LOAD_CONFIG_DIRECTORY ilcd;
GetImageConfigInformation(pLoadedImage, &ilcd);

// Change the processor affinity mask.
ilcd.ProcessAffinityMask = 0x00000003; // I desire CPUs 0 and 1

// Save the new load configuration information.
SetImageConfigInformation(pLoadedImage, &ilcd);

// Unload the EXE from memory.
ImageUnload(pLoadedImage);
```

这里不想详细说明所有这些函数的情况，如果有兴趣的话，可以在Platform SDK文档中查看这些函数的具体情况。另外，可以使用称为ImageCfg.exe的实用程序，以便改变可执行程序模块的头上的某些标志。当运行ImageCfg.exe时，它会显示下面的使用情况：

```
usage: IMAGECFG [switches] image-names...
    [-?] display this message
    [-a Process Affinity mask value in hex]
    [-b BuildNumber]
    [-c Win32 GetVersionEx Service Pack return value in hex]
    [-d decommit thresholds]
    [-g bitsToClear bitsToSet]
    [-h 1|0 (Enable/Disable Terminal Server Compatible bit)]
```

```
[ -k StackReserve[.StackCommit]
[ -l enable large (>2GB) addresses
[ -m maximum allocation size]
[ -n bind no longer allowed on this image
[ -o default critical section timeout
[ -p process heap flags]
[ -q only print config info if changed
[ -r run with restricted working set]
[ -s path to symbol files]
[ -t VirtualAlloc threshold]
[ -u Marks image as uniprocessor only]
[ -v MajorVersion.MinorVersion]
[ -w Win32 GetVersion return value in hex]
[ -x Mark image as Net - Run From Swapfile
[ -y Mark image as Removable - Run From Swapfile
```

若要修改应用程序的可允许的亲缘性屏蔽，可以执行 ImageCfg.exe 来设定 -a 开关。当然，该实用程序所做的工作只是调用上面这个代码段中显示的各个函数。还要注意的是 -u，它负责告诉系统，可执行程序只能在单个 CPU 系统上运行。

最后，Windows 2000 的 Task Manager 允许用户改变进程的 CPU 亲缘性，方法是选定一个进程，显示它的上下文菜单。如果在多处理器计算机上运行，会看到一个 Set Affinity 菜单项（该菜单项在单处理器计算机中没有）。当选择该菜单项时，会看到图 7-7 所示的对话框，在这个对话框中，可以选定进程中的线程能够在上面运行的 CPU。

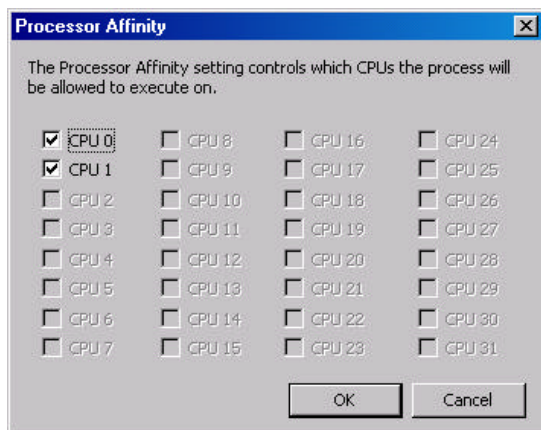


图7-7 CPU 亲缘性对话框

Windows 2000 当 Windows 2000 在 x86 计算机上引导时，可以限制系统能够使用的 CPU 的数量。在引导过程中，系统要查看称为 Boot.ini 的文件，该文件位于引导驱动器的根目录中。下面是我的双处理器计算机上的 Boot.ini 文件：

```
[boot loader]
timeout=2
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
/fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
/fastdetect /NumProcs=1
```

这个 Boot.ini 文件是 Windows 2000 安装时产生的，不过我使用 Notepad 加上了最后一行代码。这行代码告诉系统，在系统引导时，我只应该使用机器中的一个处理器。/NumProcs=1 这个开关是用来实现这一点的关键。我常常发现它对调试非常有用。

注意，由于只考虑到打印方面的需要，因此上面的程序清单中的各个选项都是在单独的一行上列出的。Boot.ini 文件要求各个选项和到达根分区的 ARC 路径必须出现在一行上。