附录四 NT 的线程机制

—在 VC、Delphi 和 Java 中多线程的实现

多线程是现代操作系统有别与传统操作系统的重要标志之一，不同与传统的多进程概念。线程也是 Windows NT 引入的先进技术之一。实际上，没有多线程，也就不会有 Windows NT 的占先式（Preemptive)多任务。线程是 Windows NT 的唯一的执行单元，Windows NT 就是靠线程的优先级及分配给 CPU 的时间来调度线程，进而达到多任务的目的的。为了是开发的应用程序可以利用线程完成特定任务，Windows NT 为开发人员提供了编程接口。但是调用 Win32 API 中的进程函数，对大多数程序开发人员来说，工作量之大是显而易见的。MFC 提供 AfxBeginThread()启动线程。在 Java 中利用创建类 Thread 的子类，并重载其 run()的方法来实现多线程;在 Delphi 中则是创建子类及其 Create()方法来实现多线程，都是面向对象编程技术的升华。本文将以 NT 为例介绍一下线程机制，并提供用 VC、Java 和 Delphi 运行环境中模拟 NT 多线程调度的程序设计思路。

一、线程的基本概念

最早应用线程这一概念的是 DEC 公司 1988 年开发的实时操作系统 VAXE'LN，此操作系统的主要设计者就是 Dave Cutler（Windows NT 的总设计师）。那时的线程概念还比较模糊，从 1989 年以后，这一概念才陆续见诸于报端，而为国内广大用户理解和接受则是在 Windows NT 发布以后。也许您已经遇见过关于线程的不同定义，例如“一个执行单元”、“一个独立的程序计数器”、“进程内的一个可调度实体”等等。很多线程的定义都是基本正确但不完善。具体来说，线程是操作系统分配 CPU 时间的最基本实体，它可以有一下几部分组成：

- 一个唯一的标志符，称为客户 ID
- 表示处理器状态的一组易变寄存器内容
- 用户态栈：线程在用户态下执行使用
- 核心态栈：线程在核心态下执行使用
- 一个私有存储区，供子系统、运行期程序库及动态链接库使用。

易变寄存器、栈和私有存储区被称为线程描述表，组成线程描述表的实际数据随处理器的不同而不同。

正如进程在逻辑上表示了操作系统所必须完成的作业一样，线程表示完成该作业的许多可能子任务之一。在 Windows NT 中，进程并不执行代码，而是有进程拥有的线程执行代码，所以线程有时又称“执行线程”。线程驻留于进程的虚拟地址空间中，在线程执行中将使用地址空间进行存储。若同一进程中存在多个线程，则它们共享地址空间和所有的处理器资源，包括其存取令牌、基本优先级以及对象表中的对象句柄。Windows NT 的内核在处理器上调度执行程序，这样一来每个进程在可执行之前必须至少具有一个线程。

注意，线程和传统操作系统常常用到的“进程”的概念是不同的。二者的差别主要体现在如下两个方面：

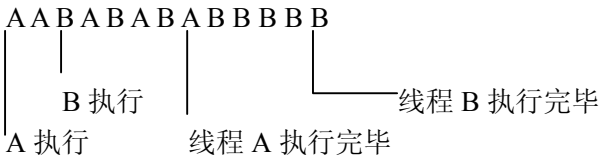
首先，同样作为基本的执行单元，线程的划分比进程小。因此，支持多线程的系统要比只支持多进程的系统并发度高。

其次，进程把内存空间作为自己的资源之一，每个资源都有的自己的内存单元。与此同时，线程却共享内存单元，通过共享的内存空间来源交换信息，从而有利提高执行效率。

二、线程的并发性和 Windows NT 的多任务

线程的概念和并发性是紧密相关的，多线程的出发点就是为了提高系统的并发性。必须注意的是，通常我们使用的计算机都是单 CPU 的，因此，所谓的并发执行，实际上从内部来看仍是串行执行的，只不过由于操作系统可自动进行任务切换，因而给人一个并发执行的假象。

比方说，如果有两个并发执行的线程 A 和 B，我们先启动线程 A 的执行，随即启动线程 B。这时，在用户看来，两个线程是并发执行的，但从系统内部来看，这两个线程却仍然串行的。如果这两个线程的优先级相同，两个线程的执行顺序将可能如下图所示：



一个处理器同一时刻只能执行一个线程，Windows NT 允许用户运行多个程序，而且看上去向是同时运行多个程序。这是通过下述方法得到的：

A.运行一个线程，直到被中断或线程必须等待到某个资源可用；

B.保存线程描述表；

C.装入另一线程描述表；

D.若存在等待被执行的线程，则重复上述过程。

多任务增加了系统所能完成的工作量，这是因为多线程不能连续地执行。线程周期性的停止执行，等待慢速 I/O 设备完成其数据传送，或等待线程所需的被其它线程占用的资源。当一个线程必须等待时，多任务方式将允许另一个线程执行，这就充分利用了本来被浪费的处理器周期。

多线程系统中，每个线程都被赋予一个优先级。优先级决定了线程获 CPU 调度执行的优先程度。优先级高的线程可在更短的时间内获得更多的执行时间，优先级低的则正好相反。线程的优先级如果完全相等，将被依据“先来先服务”的原则调度。

Windows NT 下的线程也是有优先级的，其范围从 0 到 31 共 32 级，其中 31 表示最高优先级。它们可以分成两类：实时优先级和可变化优先级。实时优先级从 16 到 31，是实时程序所用到的高优先级线程，如许多监控应用程序等。可变优先级从 0 到 15（优先级 0 为系统保留）。这些相程被成为可变优先级是因为 Windows NT 调度器为了优化系统响应时间，在它们执行过程中调整了它们的优先级。绝大多数程序的优先级都在这个范围之内。

三、线程的调度与同步

线程要能够运行，必须通过一个“调度程序”（Scheduler)来进行调度。调度实际上指的就是分配处理器资源。多线程系统中，处理器资源是按实间片分配的，获得处理器资源的线程只能在规定的实间片内执行，一旦时间片使用完毕，就必须把处理器让给另一处于等待状态的线程。

Java 和 Delphi 支持一种“抢占式”（preemptive）等待方式。抢占式是和协作式（coopeative)相对的概念。所谓协作式，是指一个执行单元一旦获得某个资源的使用权，别的执行单元就

无法剥夺，即使其它协程的优先级更高。而强制式的则相反。比方说，如果在一个低优先级线程的执行过程中，来了一个高优先级的协程，那么在协作式等待系统中，这个高优先级的线程必须等待到低优先级的相程的时间片执行完毕，而在抢占式等待方式中则不必，可以直接把控制权抢占过来。

Windows NT 就是一个抢占式多任务（preemptive multitasking)操作系统，这就意味着操作系统不必等待一个线程，它可主动将处理器让给别的线程。在这种方式下，当一个线程已经运行了预置时间后（Windows NT 中的时间大约是 20 毫秒），或者当一个高优先级的程序已经就绪运行时，操作系统将中断线程。

占先式多任务可以防止线程独占 CPU，允许其它线程公平地分享执行时间，这和 16 位 windows 下的协同多任务有着本质的区别。在 16 位 windows 环境下，如果一个程序进入无限循环，则其它应用程序可能永远没有机会执行；在 Windows NT 环境则不会，相反，许多线程的执行例程则用了无限循环。

所谓线程同步，则是指 A，B 两个线程共享一个对象时，A 和 B 线程速度的控制问题。同步的基本思想是避免多个线程访问同一个资源。Java 是纯面向对象的语言，它的资源都是以对象的形式表现的。因此，Java 的同步机制的作用力图避免对“对象”的访问冲突。Java 语言中，同步的实现在很大程度上取决于编程人员，编程人员可以决定一个类中哪些方法或代码段需同步执行，Java 提供了一种类似信号量的东西，即 Monitor，来控制对象的访问同步。同步执行的代码段要访问某个共享对象，必须拥有该对象的 Monitor。在 Delphi 中，则采用 Synchronize 的关键字来使共享对象的执行放在主线程中，从而避免多个线程的访问冲突。

四、VC 下多线程的实现

1、如何启动一个线程

用 MFC 创建一个线程，首先必须书写一个和线程对应的子程序—“线程函数”。然后用函数 AfxBeginThread()来启动线程（执行你所书写的线程函数）。如果这个线程函数运行终止，则线程相应也终止。函数 AfxBeginThread()的调用格式如下：

AfxBeginThread(ProcName, param, priority);

其中 ProcName 是指线程函数的名称，param 则是你要传递给线程的一个 32-bit 的数值，priority 是线程的优先等级，优先等级是预先定义的常数，具体定义如下表：

优先等级常数	数值意义
THREAD_PRIORITY_ABOVE_NORMAL	比正常高一级
THREAD_PRIORITY_BELOW_NORMAL	比正常低一级
THREAD_PRIORITY_HIGHEST	比正常高两级
THREAD_PRIORITY_IDLE	设置基本等级为 1
THREAD_PRIORITY_LOWEST	比正常低两级
THREAD_PRIORITY_NORMAL	正常
THREAD_PRIORITY_TIME_CRITICAL	设置基本等级为 15

注意：线程的优先等级越高，系统分配的 CPU 时间片越多。

先书写一个线程函数如下：

```
UINT ThreadProc(LPVOID param)
{
```

```
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);
    return 0;
}
```

注意：这是一个全局函数，故在函数体内不能使用 CWnd 的函数 AfxMessageBox()，而要使用 SDK 函数 MessageBox()。

以下代码将启动线程：

```
HWND hWnd = GetSafeHwnd();
AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);
```

2、主程序和线程通信

A 利用全局的控制变量

这里用一个终止线程的示例说明。如果在线程启动以后要终止它（此时线程函数还在运行），需要在程序中再添加控制变量。在程序初始化时定义一个这样的变量：

```
volatile int threadController = 0;
```

将启动线程的代码更改为：

```
threadController = 1;
HWND hWnd = GetSafeHwnd();
AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);
```

将线程函数更改为：

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);
      while (threadController == 1)
      {……}
    ::MessageBox((HWND)param, "Thread stopped.", "Thread", MB_OK);
    return 0;
}
```

终止这个线程的对应代码是：

```
threadController = 0;
```

注意：这是一种利用全局变量进行线程和主程序通信的方法，使用非常简单，但是比较危险。也不符合 Windows 的程序风格。

B 利用用户定义消息

仍然以示例的形式给出：在线程结束时给主程序发出用户消息并执行相应的子函数。具体操作步骤如下。

（1）首先定义用户消息，在头文件*类宣称*的前面加入以下代码：

```
const WM_THREADENDED = WM_USER + 100;
//定义了一个用户消息
```

（2）在头文件 *AFX_MSG* 后 *DECLARE_MESSAGE_MAP* 前加入以下代码：

```
afx_msg LONG OnThreadended(WPARAM wParam, LPARAM lParam);
```

//宣称与消息对应的执行函数

（3）在 CPP 文件*}}AFX_MSG_MAP* 后加入以下代码：

```
ON_MESSAGE(WM_THREADENDED, OnThreadended)
//将所定义的用户消息加入消息循环
```

（4）将线程函数代码更改为：

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);
        while (threadController == 1)
      {……}
    ::PostMessage((HWND)param, WM_THREADENDED, 0, 0);
    return 0;
}
```

（5）编写与消息 WM_THREADENDED 对应的函数 OnThreadended：

```
LONG CThreadView::OnThreadended(WPARAM wParam, LPARAM lParam)
{
    AfxMessageBox("Thread ended.");
    return 0;
}
```

C 利用事件对象

在 MFC 中，和事件对象对应的类是 Cevent。每个事件对象可以有两种状态：被标志和非标志，从而可以由事件对象的状态来控制线程的启动和终止。下面的示例用两个事件对象来控制线程的启动和终止。具体操作步骤如下。

（1）在 CPP 文件首部加入：

```
#include "afxmt.h"
```

（2）在 CPP 初始化部分定义：

```
CEvent threadStart;
CEvent threadEnd;
```

（3）将线程函数代码更改为：

```
UINT ThreadProc(LPVOID param)
{
    ::WaitForSingleObject(threadStart.m_hObject, INFINITE);
    BOOL keepRunning = TRUE;
    while (keepRunning)
    {
        int result =::WaitForSingleObject(threadEnd.m_hObject, 0);
        if (result == WAIT_OBJECT_0)    keepRunning = FALSE;
    }
    return 0;
}
```

（4）启动线程的相应代码更改为：

```
threadStart.SetEvent();
```

（5）终止线程的相应代码更改为：

```
threadEnd.SetEvent();
```

（6）用 ClassWizard 创建一个对应消息 WM_CREATE 的子函数 OnCreate()，内容为：

```
HWND hWnd = GetSafeHwnd();
AfxBeginThread(ThreadProc, hWnd);
```

现在应用程序运行的时候就会执行函数 OnCreate()，由于在线程函数中首行代码是 WaitForSingleObject()，所以线程暂时挂起直到事件对象 threadStart 被标志。

以上三种线程通信方法是最简单的，如果需要完成更高级的同步控制，需要用其他的方法，由于在本实验过程中只用到最简单的同步，故在此处不再介绍，由兴趣的同学请参考有关 VC 的书籍。

五、Java 多线程机制的实现

在 Java 语言中，实现多线程的方式有两种：创建类 Thread 的子类；根据 Runnable 接口创建一个类（Thread 和 Runnable 均为 Java 类库中的类）。下面我们分别给出这两种方式的 Java 程序框架。

1、创建类 Thread 的子类方式

```
class MyThread extands Thread
{   public void run()
    {    //线程的实体（后台）  }
}
class runMYThread
{   public static void main(String arg[])
    {   Thread MyThreadGet=new MyThread();
        MyThreadGet.start();
        while(MyThread.isAlive())
        {   //线程的实体（前台）  }
    }
}
```

2、根据 Runnable 接口创建一个类方式

```
class MyThread implements Runnable
{   public void run()
    {    //线程的实体（后台）  }
}
class runMyThread
{   public static void main(String arg[])
    {   Thread MyThreadGet=new MyThread();
        MyThreadGet.start();
```

```
            while(MyThread.isAlive())
            {    //线程的实体（前台）    }
        }
}
```

注意：还可以创建多个后台实体，在前台实体中激活。

六、Delphi 中多线程机制的实现

在 Delphi 中实现多线程的程序框架如下：

1.创建自己的线程类

```
TMyThread1 = class(TThread)
  private
  protected
  public
    constructor Create();
  end;
{创建自己的线程 Create()方法}
constructor TMyThread1.Create();
begin
    {后台线程的实体}
  FreeOnTerminate := True;
  inherited Create(False);
end;
```

2.在前台的线程实体中运行后台的线程

```
{前台线程的实体}
{前台调用线程}
TMyThread1.Create();
{前台线程的实体}
```

七、参考文献

【1】 Beck Zaratian 著，希望图书创作室译，Microsoft Visual C++ 6.0 程序员指南〔美〕，北京：北京希望电脑公司，1998

【2】 “http://202.38.64.40/books/c/sevc++5/index27.htm”，Chapter 27 Of “Special Edition Using Visual C++ 5”

【3】 Beck Zaratian 著，希望图书创作室译，Microsoft Visual C++ 6.0 程序员指南〔美〕，北京：北京希望电脑公司，1998

【4】 沈浩 冯磊 肖骏，Win32 环境下 Socket、Multithread 编程初探，新浪潮，1998.5

【5】 Delphi 3.0 联机帮助

【6】 Visual C++ 5.0 联机帮助