5 实验二 并发与调度

[1]Windows Server 2003 线程同步

背景知识

Windows Server 2003 提供的常用对象可分成三类：核心应用服务、线程同步和线程间通讯。其中，开发人员可以使用线程同步对象来协调线程和进程的工作，以使其共享信息并执行任务。此类对象包括互锁数据、临界段、事件、互斥体和信号等。

多线程编程中关键的一步是保护所有的共享资源，工具主要有互锁函数、临界段和互斥体等；另一个实质性部分是协调线程使其完成应用程序的任务，为此，可利用内核中的事件对象和信号。

在进程内或进程间实现线程同步的最方便的方法是使用事件对象，这一组内核对象允许一个线程对其受信状态进行直接控制 (见表 4-1) 。

而互斥体则是另一个可命名且安全的内核对象，其主要目的是引导对共享资源的访问。拥有单一访问资源的线程创建互斥体，所有想要访问该资源的线程应该在实际执行操作之前获得互斥体，而在访问结束时立即释放互斥体，以允许下一个等待线程获得互斥体，然后接着进行下去。

与事件对象类似，互斥体容易创建、打开、使用并清除。利用 CreateMutex() API 可创建互斥体，创建时还可以指定一个初始的拥有权标志，通过使用这个标志，只有当线程完成了资源的所有的初始化工作时，才允许创建线程释放互斥体。

表 4-1 用于管理事件对象的 API

API 名称	描述
CreateEvent()	在内核中创建一个新的事件对象。此函数允许有安全性设置、手工还是自动重置的标志以及初始时已接受还是未接受信号状态的标志
OpenEvent()	创建对已经存在的事件对象的引用。此 API 函数需要名称、继承标志和所需的访问级别
SetEvent()	将手工重置事件转化为已接受信号状态
ResetEvent()	将手工重置事件转化为非接受信号状态
PulseEvent()	将自动重置事件对象转化为已接受信号状态。当系统释放所有的等待它的线程时此种转化立即发生

为了获得互斥体，首先，想要访问调用的线程可使用 OpenMutex() API 来获得指向对象的句柄；然后，线程将这个句柄提供给一个等待函数。当内核将互斥体对象发送给等待线程时，就表明该线程获得了互斥体的拥有权。当线程获得拥有权时，线程控制了对共享资源的访问——必须设法尽快地放弃互斥体。放弃共享资源时需要在该对象上调用 ReleaseMute() API。然后系统负责将互斥体拥有权传递给下一个等待着的线程 (由到达时间决定顺序) 。

实验目的

在本实验中，通过对事件和互斥体对象的了解，来加深对 Windows Server 2003 线程同步的理解。

1) 回顾系统进程、线程的有关概念，加深对 Windows Server 2003 线程的理解。
2) 了解事件和互斥体对象。
3) 通过分析实验程序，了解管理事件对象的 API。
4) 了解在进程中如何使用事件对象。
5) 了解在进程中如何使用互斥体对象。

6) 了解父进程创建子进程的程序设计方法。

工具/准备工作

在开始本实验之前，请回顾教科书的相关内容。
您需要做以下准备：
1) 一台运行 Windows Server 2003 操作系统的计算机。
2) 计算机中需安装 Visual C++ 6.0 专业版或企业版。

实验内容与步骤

1. 事件对象

清单 4-1 程序展示了如何在进程间使用事件。父进程启动时，利用 CreateEvent() API 创建一个命名的、可共享的事件和子进程，然后等待子进程向事件发出信号并终止父进程。在创建时，子进程通过 OpenEvent() API 打开事件对象，调用 SetEvent() API 使其转化为已接受信号状态。两个进程在发出信号之后几乎立即终止。

步骤 1：登录进入 Windows Server 2003 。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio 6.0”–“Microsoft Visual C++ 6.0”命令，进入 Visual C++窗口。

步骤 3：在工具栏单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 4-1.cpp。

清单 4-1 创建和打开事件对象在进程间传送信号

```
// event 项目
# include <windows.h>
# include <iostream>

// 以下是句柄事件。实际中很可能使用共享的包含文件来进行通讯
static LPCTSTR g_szContinueEvent = "w2kdg.EventDemo.event.Continue";

// 本方法只是创建了一个进程的副本，以子进程模式 (由命令行指定) 工作
BOOL CreateChild()
{
    // 提取当前可执行文件的文件名
        TCHAR szFilename[MAX_PATH] ;
    :: GetModuleFileName(NULL, szFilename, MAX_PATH) ;

        // 格式化用于子进程的命令行，指明它是一个 EXE 文件和子进程
    TCHAR szCmdLine[MAX_PATH] ;
    :: sprintf(szCmdLine, "\"%s\"child" , szFilename) ;

    // 子进程的启动信息结构
    STARTUPINFO si;
    :: ZeroMemory(reinterpret_cast<void*>(&si), sizeof(si)) ;
    si.cb = sizeof(si);             // 必须是本结构的大小

    // 返回的子进程的进程信息结构
    PROCESS_INFORMATION pi;

    // 使用同一可执行文件和告诉它是一个子进程的命令行创建进程
    BOOL bCreateOK = :: CreateProcess(
        szFilename,             // 生成的可执行文件名
        szCmdLine,                  // 指示其行为与子进程一样的标志
```

```
        NULL,                   // 子进程句柄的安全性
        NULL,                   // 子线程句柄的安全性
        FALSE,                      // 不继承句柄
        0,                      // 特殊的创建标志
        NULL,                   // 新环境
        NULL,                   // 当前目录
        &si,                        // 启动信息结构
        &pi ) ;                 // 返回的进程信息结构

        // 释放对子进程的引用
        if (bCreateOK)
            {
             :: CloseHandle(pi.hProcess);
             :: CloseHandle(pi.hThread);
        }
        return(bCreateOK) ;
    }

    // 下面的方法创建一个事件和一个子进程，然后等待子进程在返回前向事件发出信号
    void WaitForChild()
    {
        // create a new event object for the child process
        // to use when releasing this process
        HANDLE hEventContinue = :: CreateEvent(
            NULL,                   // 缺省的安全性，子进程将具有访问权限
            TRUE,                   // 手工重置事件
            FALSE,                      // 初始时是非接受信号状态
            g_szContinueEvent);         // 事件名称
        if (hEventContinue!=NULL)
        {
                std :: cout << "event created " << std :: endl;

                // 创建子进程
            if (:: CreateChild())
            {
                 std :: cout << " chlid created" << std :: endl;

                 // 等待，直到子进程发出信号
                 std :: cout << "Parent waiting on child." << std :: endl;
                 :: WaitForSingleObject(hEventContinue, INFINITE);

                 :: Sleep(1500);                    // 删去这句试试
                    std :: cout << "parent received the envent signaling from child" << std ::
endl;
            }

            // 清除句柄
            :: CloseHandle(hEventContinue);
            hEventContinue = INVALID_HANDLE_VALUE;
        }
    }

    // 以下方法在子进程模式下被调用，其功能只是向父进程发出终止信号
    void SignalParent()
    {
```

```
        // 尝试打开句柄
            std :: cout << "child process begining....." << std :: endl;
        HANDLE hEventContinue = :: OpenEvent(
              EVENT_MODIFY_STATE,          // 所要求的最小访问权限
              FALSE,                       // 不是可继承的句柄
        g_szContinueEvent);                // 事件名称
        if (hEventContinue != NULL)
            {
             :: SetEvent(hEventContinue);
                std :: cout << "event signaled" << std :: endl;
        }

        // 清除句柄
        :: CloseHandle(hEventContinue) ;
        hEventContinue = INVALID_HANDLE_VALUE;
    }

    int main(int argc, char* argv[] )
    {
        // 检查父进程或是子进程是否启动
        if (argc>1 && :: strcmp(argv[1] , "child" )== 0)
        {
            // 向父进程创建的事件发出信号
             :: SignalParent() ;
        }
        else
        {
                  // 创建一个事件并等待子进程发出信号
             :: WaitForChild() ;
                 :: Sleep(1500) ;
             std :: cout << "Parent released." << std :: endl ;
        }
        return 0;
    }
```

步骤 4：单击“Build”菜单中的“Compile 4-1.cpp”命令，并单击“是”按钮确认。系统对 4-1.cpp 进行编译。

步骤 5：编译完成后，单击“Build”菜单中的“Build 4-1.exe”命令，建立 4-1.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 6：在工具栏单击“Execute Program” (执行程序) 按钮，执行 4-1.exe 程序。

运行结果 (分行书写。如果运行不成功，则可能的原因是什么？) ：

1) _____

2) _____

3) _____

4) _____

5) _____

6) _____

这个结果与你期望的一致吗？(从进程并发的角度对结果进行分析)

阅读和分析程序 4-1，请回答：

1) 程序中，创建一个事件使用了哪一个系统函数？创建时设置的初始信号状态是什么？

   a. _____

   b. _____

2) 创建一个进程 (子进程) 使用了哪一个系统函数？

_____

3) 从步骤 6 的输出结果，对照分析 4-1 程序，可以看出程序运行的流程吗？请简单描述：

_____
_____
_____
_____
_____
_____

2. 互斥体对象

清单 4-2 的程序中显示的类 CCountUpDown 使用了一个互斥体来保证对两个线程间单一数值的访问。每个线程都企图获得控制权来改变该数值，然后将该数值写入输出流中。创建者实际上创建的是互斥体对象，计数方法执行等待并释放，为的是共同使用互斥体所需的资源 (因而也就是共享资源) 。

步骤 7：在 Visual C++ 窗口的工具栏中单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 4-2.cpp。

清单 4-2　利用互斥体保护共享资源

```
// mutex 项目
# include <windows.h>
# include <iostream>
// 利用互斥体来保护同时访问的共享资源
class CCountUpDown
{
  public:
    // 创建者创建两个线程来访问共享值
    CCountUpDown(int nAccesses) :
      m_hThreadInc(INVALID_HANDLE_VALUE) ,
      m_hThreadDec(INVALID_HANDLE_VALUE) ,
      m_hMutexValue(INVALID_HANDLE_VALUE) ,
      m_nValue(0) ,
      m_nAccess(nAccesses)
    {
      // 创建互斥体用于访问数值
      m_hMutexValue = :: CreateMutex(
          NULL,              // 缺省的安全性
          TRUE,              // 初始时拥有，在所有的初始化结束时将释放
          NULL) ;            // 匿名的
      m_hThreadInc = :: CreateThread(
          NULL,              // 缺省的安全性
          0,                 // 缺省堆栈
          IncThreadProc,     // 类线程进程
          reinterpret_cast <LPVOID> (this) ,          // 线程参数
          0,                 // 无特殊的标志
          NULL) ;            // 忽略返回的 id
      m_hThreadDec = :: CreateThread(
          NULL,              // 缺省的安全性
          0,                 // 缺省堆栈
          DecThreadProc,     // 类线程进程
          reinterpret_cast <LPVOID> (this) ,          // 线程参数
          0,                 // 无特殊的标志
          NULL) ;            // 忽略返回的 id
```

```
        // 允许另一线程获得互斥体
        :: ReleaseMutex(m_hMutexValue) ;
    }

    // 解除程序释放对对象的引用
virtual ~CCountUpDown()
    {
        :: CloseHandle(m_hThreadInc) ;
        :: CloseHandle(m_hThreadDec) ;
        :: CloseHandle(m_hMutexValue) ;
    }

    // 简单的等待方法，在两个线程终止之前可暂停主调者
    virtual void WaitForCompletion()
    {
        // 确保所有对象都已准备好
        if (m_hThreadInc != INVALID_HANDLE_VALUE &&
            m_hThreadDec != INVALID_HANDLE_VALUE)
        {
            // 等待两者完成 (顺序并不重要)
            :: WaitForSingleObject(m_hThreadInc, INFINITE) ;
            :: WaitForSingleObject(m_hThreadDec, INFINITE) ;
        }
    }

protected:
    // 改变共享资源的简单的方法
    virtual void DoCount(int nStep)
    {
        // 循环，直到所有的访问都结束为止
        while (m_nAccess > 0)
        {
            // 等待访问数值
            :: WaitForSingleObject(m_hMutexValue, INFINITE) ;

            // 改变并显示该值
            m_nValue += nStep;

            std :: cout << "thread: " << :: GetCurrentThreadId()
                        << "value: " << m_nValue
                        << "access: " << m_nAccess << std :: endl;

            // 发出访问信号并允许线程切换
            --m_nAccess;
            :: Sleep(1000) ;                // 使显示速度放慢

            // 释放对数值的访问
            :: ReleaseMutex(m_hMutexValue) ;
        }
    }
    static DWORD WINAPI IncThreadProc(LPVOID lpParam)
    {
        // 将参数解释为 'this' 指针
        CCountUpDown* pThis =
            reinterpret_cast < CCountUpDown* > (lpParam) ;
```

```
 // 调用对象的增加方法并返回一个值
                pThis -> DoCount(+1) ;
                return(0) ;
            }

            static DWORD WINAPI DecThreadProc(LPVOID lpParam)
            {
                // 将参数解释为 'this' 指针
                CCountUpDown* pThis =
                   reinterpret_cast <CCountUpDown* > (lpParam) ;
                // 调用对象的减少方法并返回一个值
                pThis -> DoCount(-1) ;
                return(0) ;
            }

protected:
            HANDLE m_hThreadInc;
            HANDLE m_hThreadDec;

            HANDLE m_hMutexValue;
            int m_nValue;
            int m_nAccess ;
} ;

        void main()
        {
            CCountUpDown ud(50) ;
            ud.WaitForCompletion() ;
        }
```

步骤 8：单击“Build”菜单中的“Compile 4-2.cpp”命令，并单击“是”按钮确认。系统对 4-2.cpp 进行编译。

步骤 9：编译完成后，单击“Build”菜单中的“Build 4-2.exe”命令，建立 4-2.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 10：在工具栏单击“Execute Program”按钮，执行 4-2.exe 程序。

分析程序 4-2 的运行结果，可以看到线程 (加和减线程) 的交替执行 (因为 Sleep() API 允许 Windows 切换线程) 。在每次运行之后，数值应该返回初始值 (0) ，因为在每次运行之后写入线程在等待队列中变成最后一个，内核保证它在其他线程工作时不会再运行。

1) 请描述运行结果 (如果运行不成功，则可能的原因是什么？) ：

_____

_____

_____

_____

_____

2) 根据运行输出结果，对照分析 4-2 程序，可以看出程序运行的流程吗？请简单描述：

_____

_____

_____

_____

_____

_____