第2章 Windows 的进程管理

2.1 实验一：线程的创建与撤销

2.1.1 实验目的

(1) 熟悉 Windows 系统提供的线程创建与撤销系统调用。

(2) 掌握 Windows 系统环境下线程的创建与撤销方法。

2.1.2 实验准备知识：相关 API 函数介绍

1. 线程创建

CreateThread()完成线程的创建。它在调用进程的地址空间上创建一个线程，执行指定的函数，并返回新建立线程的句柄。

原型：

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES lpThreadAttributes,      // 安全属性指针
 DWORD dwStackSize,                             // 线程堆栈大小
 LPTHREAD_START_ROUTINE lpStartAddress,         // 线程所要执行的函数
 LPVOID lpParameter,                            // 线程对应函数要传递的参数
 DWORD dwCreationFlags,                         // 线程创建后所处的状态
 LPDWORD lpThreadId                             // 线程标识符指针
);
```

参数说明：

(1) lpThreadAttributes：为线程指定安全属性。为 NULL 时，线程得到一个默认的安全描述符。

(2) dwStackSize：线程堆栈的大小。其值为 0 时，其大小与调用该线程的线程堆栈大小相同。

(3) lpStartAddress：指定线程要执行的函数。

(4) lpParameter：函数中要传递的参数。

(5) dwCreationFlags：指定线程创建后所处的状态。若为 CREATE_SUSPENDED，表示创建后处于挂起状态，用 ResumeThread()激活后线程才可执行。若该值设为 0，表示线程创建后立即执行。

(6) lpThreadId：用一个 32 位的变量接收系统返回的线程标识符。若该值设为 NULL，系统不返回线程标识符。

返回值：

如果线程创建成功，将返回该线程的句柄；如果失败，系统返回 NULL，可以调用函数 GetLastError 查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;                    // 用于存储线程返回句柄的变量
DWORD dwThreadID1;                                 // 用于存储线程标识符的变量
// 创建一个名为 ThreadName1 的线程
hHandle1 = CreateThread((LPSECURITY_ATTRIBUTES) NULL,
                        0,
                        (LPTHREAD_START_ROUTINE) ThreadName1,
                        (LPVOID) NULL,
                        0, &dwThreadID1);
```

2. 撤销线程

ExitThread()用于撤销当前线程。

原型：

```
VOID ExitThread(
 DWORD dwExitCode                                  // 线程返回码
);
```

参数说明：

dwExitCode：指定线程返回码，可以调用 GetExitCodeThread()查询返回码的含义。

返回值：

该函数没有返回值。

用法举例：

```
ExitThread(0);                                     // 参数 0 表示要撤销进程中的所有线程
```

3. 终止线程

TerminateThread()用于终止当前线程。该函数与 ExitThread()的区别在于，ExitThread()在撤销线程时将该线程所拥有的资源全部归还给系统，而 TerminateThread()不归还资源。

原型：

```
BOOL TerminateThread(
 HANDLE hThread,                                   // 线程句柄
 DWORD dwExitCode                                  // 线程返回码
);
```

参数说明：

(1) hThread：要终止线程的线程句柄。

(2) dwExitCode：指定线程返回码，可以调用 GetExitCodeThread()查询返回码的含义。

返回值：

函数调用成功，将返回一个非 0 值；若失败，返回 0，可以调用函数 GetLastError()查询失败的原因。

4. 挂起线程

Sleep()用于挂起当前正在执行的线程。

原型：

```
VOID Sleep(
 DWORD dwMilliseconds                    // 挂起时间
);
```

参数说明：

dwMilliseconds：指定挂起时间，单位为 ms(毫秒)。

返回值：

该函数无返回值。

5. 关闭句柄

函数 CloseHandle()用于关闭已打开对象的句柄，其作用与释放动态申请的内存空间类似，这样可以释放系统资源，使进程安全运行。

原型：

```
BOOL CloseHandle(
 HANDLE hObject                          // 要关闭对象的句柄
);
```

参数说明：

hObject：已打开对象的句柄。

返回值：

如果函数调用成功，则返回值为非 0 值；如果函数调用失败，则返回值为 0。若要得到更多的错误信息，调用函数 GetLastError()查询。

2.1.3 实验内容

使用系统调用 CreateThread()创建一个子线程，并在子线程序中显示：Thread is Runing!。为了能让用户清楚地看到线程的运行情况，使用 Sleep()使线程挂起 5s，之后使用 ExitThread(0)撤销线程。

2.1.4 实验要求

能正确使用 CreateThread()、ExitThread()及 Sleep()等系统调用，进一步理解进程与线程理论。

2.1.5 实验指导

本实验在 Windows XP、Microsoft Visual C++ 6.0 环境下实现，利用 Windows SDK (System Development Kit)提供的 API(Application Program Interface，应用程序接口)完成程序的功能。实验在 Windows XP 环境下安装 Microsoft Visual C++ 6.0 后进行，由于 Microsoft Visual C++ 6.0 是一个集成开发环境，其中包含了 Windows SDK 所有工具和定义，所以安装了 Microsoft Visual C++ 6.0 后不用特意安装 SDK。实验中所有的 API 是操作系统提供的用来进行应用程序开发的系统功能接口。

(1) 首先启动安装好的 Microsoft Visual C++ 6.0。

(2) 在 Microsoft Visual C++ 6.0 环境下选择 File→New 命令，然后在 Project 选项卡中选择 Win32 Console Application 建立一个控制台工程文件。

(3) 由于 CreateThread()等函数是 Microsoft Windows 操作系统的系统调用，因此在图 2-1 中选择 An application that supports MFC，之后单击 Finish 按钮。

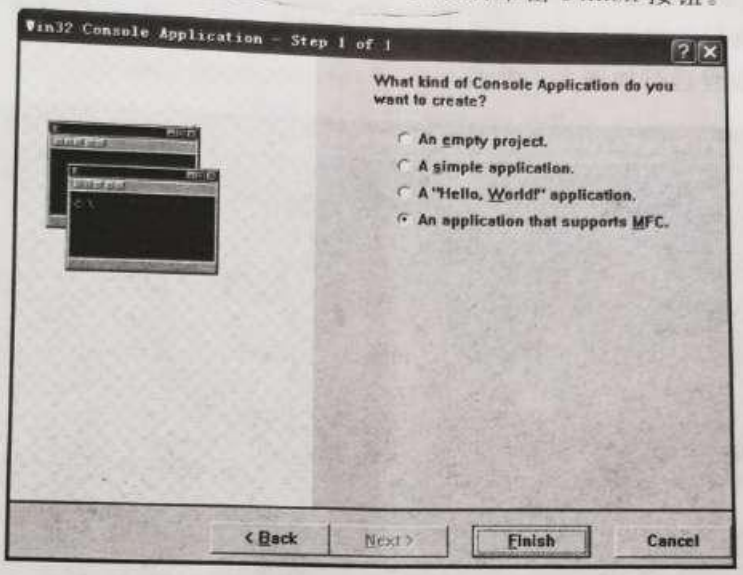

图 2-1 建立一个 MFC 支持的应用程序

(4) 之后将打开 Microsoft Visual C++ 6.0 编辑环境(见图 2-2)，按本实验的要求编辑 C 程序，之后编译、链接并运行该程序即可。

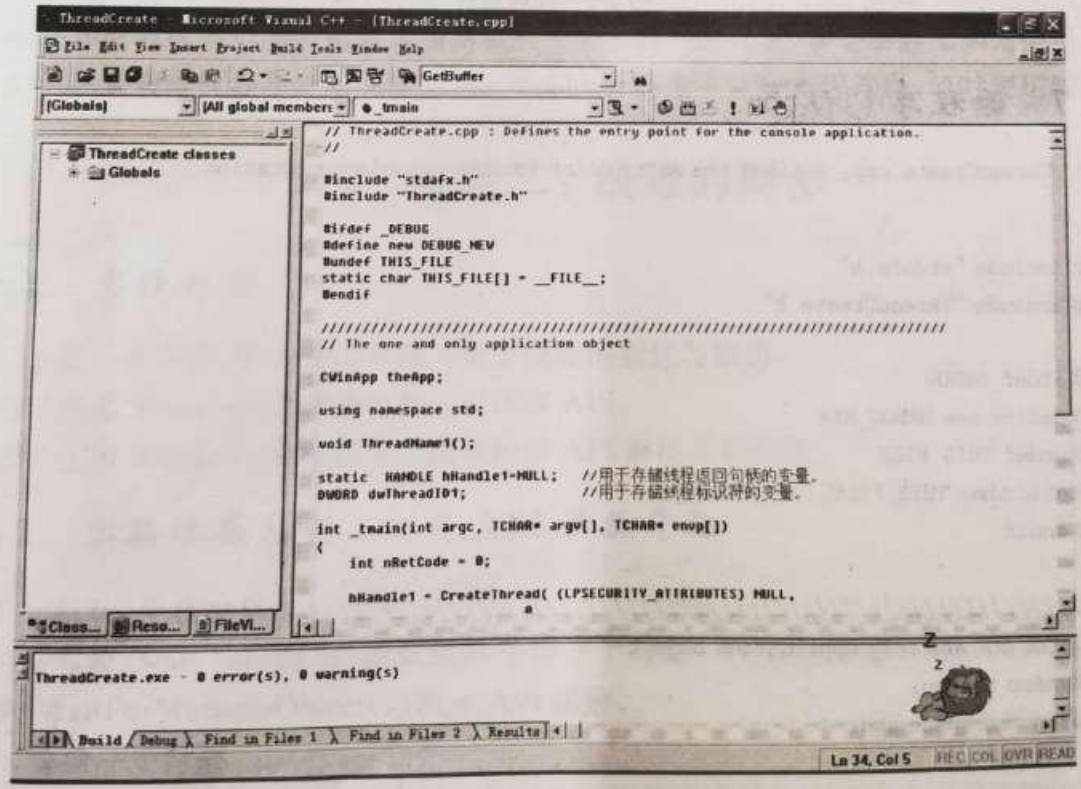

图 2-2 Microsoft Visual C++ 6.0 编辑环境

2.1.6 实验总结

在 Windows 系统中进程是资源的拥有者，线程是系统调度的单位。进程创建后，其主线程也随即被创建。在该实验中，又创建了一个名为 ThreadName1 的子线程，该子线程与主线程并发地被系统调度。为了能看到子线程的运行情况，在主线程创建了子线程后，将主线程挂起 5s 以确保子线程能够运行完毕，之后调用 ExitThread(0)将所有线程(包括主、子线程序)撤销。线程运行如图 2-3 所示。

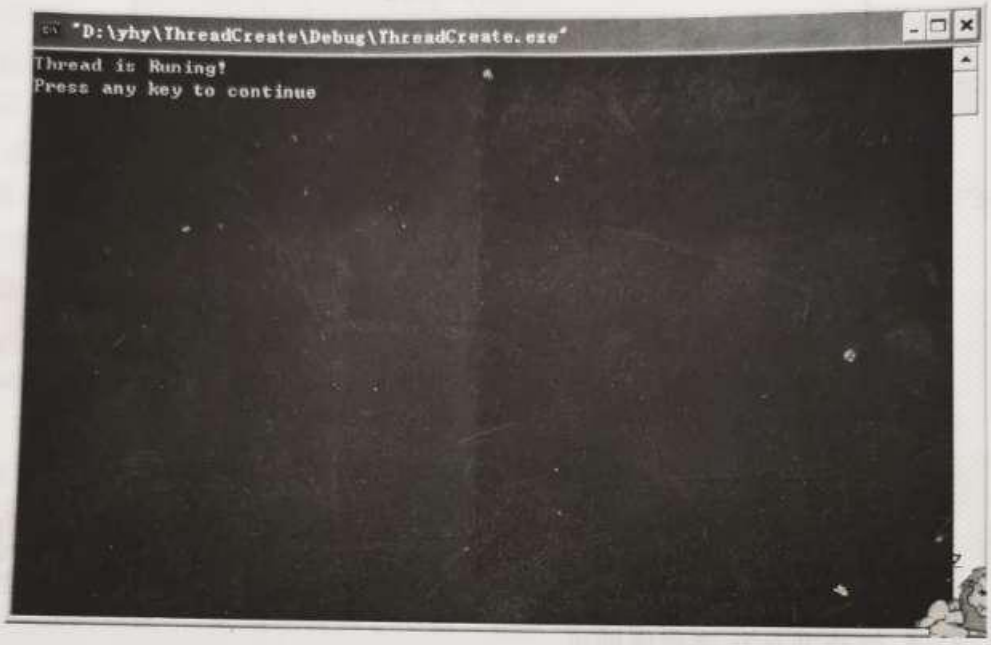

图 2-3 线程运行

2.1.7 源程序(附录)

```
// ThreadCreate.cpp: Defines the entry point for the console application.

#include "stdafx.h"
#include "ThreadCreate.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////////
// The one and only application object
CWinApp theApp;
using namespace std;

void ThreadName1();
```

```
static   HANDLE hHandle1 = NULL;                    //用于存储线程返回句柄的变量
DWORD dwThreadID1;                                  //用于存储线程标识符的变量

int_tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int nRetCode = 0;
    hHandle1 = CreateThread((LPSECURITY_ATTRIBUTES) NULL,
                0,
                (LPTHREAD_START_ROUTINE) ThreadName1,
                        //创建一个名 ThreadName1 的线程
                (LPVOID) NULL,
                0,
                &dwThreadID1);
    Sleep(5000);                                    //将主线程挂起 5s
    CloseHandle(hHandle1);                          //关闭句柄
    ExitThread(0);                                  //撤销线程

    return nRetCode;
}

void ThreadName1()                                  //线程对应的函数
{
    printf("Thread is Runing!\n");
}
```

2.1.8 实验展望

可以进一步完善程序功能，请思考以下问题。

(1) 如何向线程对应的函数传递参数？一个参数如何传递，多个参数又如何传递？

(2) 深入理解线程与进程的概念，在 Windows 环境下何时使用进程，何时使用线程？

2.2 实验二：线程的同步

2.2.1 实验目的

(1) 进一步掌握 Windows 系统环境下线程的创建与撤销。

(2) 熟悉 Windows 系统提供的线程同步 API。

(3) 使用 Windows 系统提供的线程同步 API 解决实际问题。

2.2.2 实验准备知识：相关 API 函数介绍

2.2.2.1 等待对象

等待对象(wait functions)函数包括等待一个对象(WaitForSingleObject())和等待多个对象(WaitForMultipleObject())两个 API 函数。

1. 等待一个对象

WaitForSingleObject()用于等待一个对象。它等待的对象可以为以下对象之一。

- Change notification：变化通知。
- Console input：控制台输入。
- Event：事件。
- Job：作业。
- Mutex：互斥信号量。
- Process：进程。
- Semaphore：计数信号量。
- Thread：线程。
- Waitable timer：定时器。

原型：

```
DWORD WaitForSingleObject(
 HANDLE hHandle,                    // 对象句柄
 DWORD dwMilliseconds               // 等待时间
);
```

参数说明：

(1) hHandle：等待对象的对象句柄。该对象句柄必须为 SYNCHRONIZE 访问。

(2) dwMilliseconds：等待时间，单位为 ms。若该值为 0，函数在测试对象的状态后立即返回，若为 INFINITE，函数一直等待下去，直到接收到一个信号将其唤醒，如表 2-1 所示。

返回值：

如果成功返回，其返回值说明是何种事件导致函数返回。

表 2-1 函数描述

访　问	描　述
WAIT_ABANDONED	等待的对象是一个互斥(Mutex)对象，该互斥对象没有被拥有它的线程释放，它被设置为不能被唤醒
WAIT_OBJECT_0	指定对象被唤醒
WAIT_TIMEOUT	超时

用法举例：

```
static  HANDLE hHandle1 = NULL;
DWORD  dRes;
dRes = WaitForSingleObject(hHandle1,10);    //等待对象的句柄为 hHandle1,等待时间为 10ms
```

2. 等待多个对象

WaitForMultipleObject( ) 在指定时间内等待多个对象，它等待的对象与 WaitForSingleObject()相同。

原型：

```
DWORD WaitForMultipleObjects(
  DWORD nCount,                     // 句柄数组中的句柄数
  CONST HANDLE * lpHandles,         // 指向对象句柄数组的指针
```

```
    BOOL fWaitAll,                          // 等待类型
    DWORD dwMilliseconds                    // 等待时间
);
```

参数说明：

(1) nCount：由指针 * lpHandles 指定的句柄数组中的句柄数，最大数是 MAXIMUM_WAIT_OBJECTS。

(2) * lpHandles：指向对象句柄数组的指针。

(3) fWaitAll：等待类型。若为 TRUE，当由 lpHandles 数组指定的所有对象被唤醒时函数返回；若为 FALSE，当由 lpHandles 数组指定的某一个对象被唤醒时函数返回，且由返回值说明是由于哪个对象引起的函数返回。

(4) dwMilliseconds：等待时间，单位为 ms。若该值为 0，函数测试对象的状态后立即返回；若为 INFINITE，函数一直等待下去，直到接收到一个信号将其唤醒。

返回值：

如果成功返回，其返回值说明是何种事件导致函数返回。

各参数的描述如表 2-2 所示。

表 2-2 各参数描述

访 问	描 述
WAIT_OBJECT_0 to (WAIT_OBJECT_0+nCount-1)	若 bWaitAll 为 TRUE，返回值说明所有被等待的对象均被唤醒；若 bWaitAll 为 FALSE，返回值减去 WAIT_OBJECT_0 说明 lpHandles 数组下标指定的对象满足等待条件。如果调用时多个对象同时被唤醒，则取多个对象中最小的那个数组下标
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0+nCount-1)	若 bWaitAll 为 TRUE，返回值说明所有被等待的对象均被唤醒，并且至少有一个对象是没有约束的互斥对象；若 bWaitAll 为 FALSE，返回值减去 WAIT_ABANDONED_0 说明 lpHandles 数组下标指定的没有约束的互斥对象满足等待条件
WAIT_TIMEOUT	超时且参数 bWaitAll 指定的条件不能满足

2.2.2.2 信号量对象(Semaphore)

信号量对象(Semaphore)包括创建信号量(CreateSemaphore())、打开信号量(OpenSemaphore())及增加信号量的值(ReleaseSemaphore())API 函数。

1. 创建信号量

CreateSemaphore()用于创建一个信号量。

原型：

```
HANDLE CreateSemaphore(
 LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,     // 安全属性
 LONG lInitialCount,                              // 信号量对象的初始值
 LONG lMaximumCount,                              // 信号量的最大值
 LPCTSTR lpName                                   // 信号量名
);
```

参数说明：

(1) lpSemaphoreAttributes：指定安全属性，为 NULL 时，信号量得到一个默认的安全

描述符。

(2) lInitialCount：指定信号量对象的初始值。该值必须大于等于0，小于等于lMaximumCount。当其值大于0时，信号量被唤醒。当该函数释放了一个等待该信号量的线程时，lInitialCount值减1，当调用函数ReleaseSemaphore()时，按其指定的数量加一个值。

(3) lMaximumCount：指出该信号量的最大值，该值必须大于0。

(4) lpName：给出信号量的名字。

返回值：

信号量创建成功，将返回该信号量的句柄。如果给出的信号量名是系统已经存在的信号量，将返回这个已存在信号量的句柄。如果失败，系统返回NULL，可以调用函数GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;                          //定义一个句柄
//创建一个信号量，其初值为0，最大值为5，信号量的名字为"SemphoreName1"
hHandle1 = CreateSemaphore(NULL,0,5,"SemphoreName1");
```

2. 打开信号量

OpenSemaphore()用于打开一个信号量。

原型：

```
HANDLE OpenSemaphore(
 DWORD dwDesiredAccess,                    // 访问标志
 BOOL bInheritHandle,                      // 继承标志
 LPCTSTR lpName                            // 信号量名
);
```

参数说明：

(1) dwDesiredAccess：指出打开后要对信号量进行何种访问，如表2-3所示。

表2-3 访问状态

访　问	描　述
SEMAPHORE_ALL_ACCESS	可以进行任何对信号量的访问
SEMAPHORE_MODIFY_STATE	可使用ReleaseSemaphore()修改信号量的值，使信号量成为可用状态
SYNCHRONIZE	使用等待函数(wait functions)，等待信号量成为可用状态

(2) bInheritHandle：指出返回的信号量句柄是否可以继承。

(3) lpName：给出信号量的名字。

返回值：

信号量打开成功，将返回该信号量的句柄；如果失败，系统返回NULL，可以调用函数GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;
```

```
// 打开一个名为"SemphoreName1"的信号量，之后可使用 ReleaseSemaphore()函数增加信号量的值
hHandle1 = OpenSemaphore(SEMAPHORE_MODIFY_STATE,NULL,"SemphoreName1");
```

3. 增加信号量的值

ReleaseSemaphore()用于增加信号量的值。

原型：

```
BOOL ReleaseSemaphore(
 HANDLE hSemaphore,                    // 信号量对象句柄
 LONG lReleaseCount,                   // 信号量要增加数值
 LPLONG lpPreviousCount                // 信号量要增加数值的地址
);
```

参数说明：

(1) hSemaphore：创建或打开信号量时给出的信号量对象句柄。Windows NT 中建议要使用 SEMAPHORE_MODIFY_STATE 访问属性打开该信号量。

(2) lReleaseCount：信号量要增加的数值。该值必须大于 0。如果增加该值后，大于信号量创建时给出的 lMaximumCount 值，则增加操作失效，函数返回 FALSE。

(3) lpPreviousCounte：接收信号量值的一个 32 位的变量。若不需要接收该值，可以指定为 NULL。

返回值：

如果成功，将返回一个非 0 值；如果失败，系统返回 0，可以调用函数 GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;
BOOL rc;
rc = ReleaseSemaphore(hHandle1,1,NULL);          // 给信号量的值加 1
```

2.2.3 实验内容

完成主、子两个线程之间的同步，要求子线程先执行。在主线程中使用系统调用 CreateThread()创建一个子线程。主线程创建子线程后进入阻塞状态，直到子线程运行完毕后唤醒主线程。

2.2.4 实验要求

能正确使用等待对象 WaitForSingleObject()或 WaitForMultipleObject()及信号量对象 CreateSemaphore()、OpenSemaphore()、ReleaseSemaphore()等系统调用，进一步理解线程的同步。

2.2.5 实验指导

具体操作过程同本章实验一，在 Microsoft Visual C++ 6.0 环境下建立一个 MFC 支持的控制台工程文件，编写 C 程序，在程序中使用 CreateSemaphore (NULL, 0, 1, "SemaphoreName1")创建一个名为“SemaphoreName1”的信号量，信号量的初始值为 0，之

后使用 OpenSemaphore(SYNCHRONIZE | SEMAPHORE_MODIFY_STATE, NULL, "SemaphoreName1")打开该信号量,这里访问标志用"SYNCHRONIZE | SEMAPHORE_MODIFY_STATE",以便之后可以使用 WaitForSingleObject()等待该信号量及使用 ReleaseSemaphore()释放该信号量,然后创建一个子线程,主线程创建子线程后调用 WaitForSingleObject(hHandle1,INFINITE),这里等待时间设置为 INFINITE 表示要一直等待下去,直到该信号量被唤醒为止。子线程结束,调用 ReleaseSemaphore(hHandle1,1,NULL)释放信号量,使信号量的值加 1。

2.2.6 实验总结

该实验完成了主、子两个线程的同步,主线程创建子线程后,主线程阻塞,让子线程先执行,等子线程执行完毕后,由子线程唤醒主线程。主、子线程运行情况如图 2-4 所示。

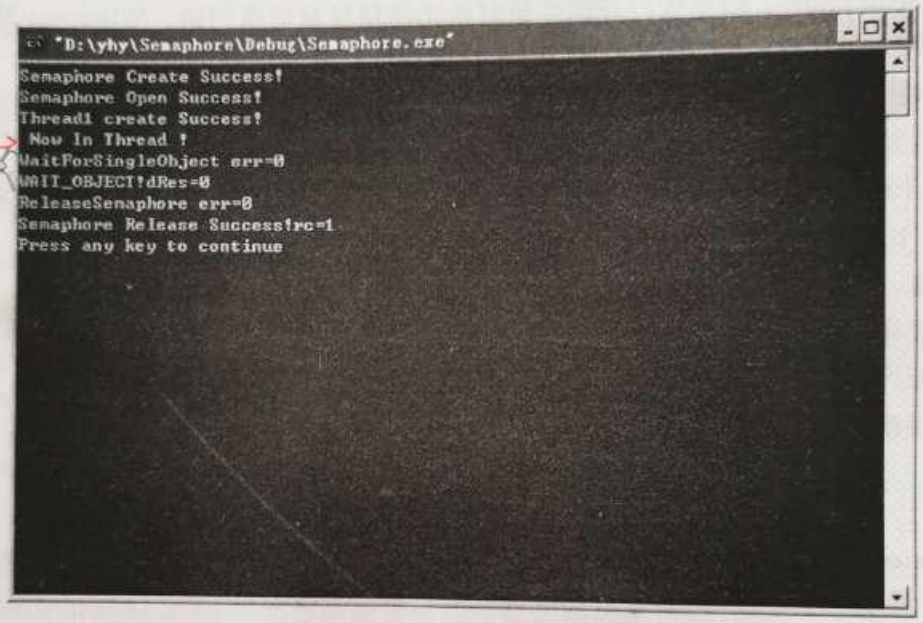

图 2-4 主、子线程运行情况

2.2.7 源程序

```
// Semaphore.cpp: Defines the entry point for the console application.
//
#include "stdafx.h"
#include "Semaphore.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////
// The one and only application object
```

```
CWinApp theApp;
using namespace std;

static HANDLE h1;                                    //线程句柄
static HANDLE hHandle1 = NULL;                       //信号量句柄

void func();

int _tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int nRetCode = 0;
    DWORD  dwThreadID1;
    DWORD dRes,err;

    hHandle1 = CreateSemaphore(NULL,0,1,"SemaphoreName1");    //创建一个信号量
    if   (hHandle1 == NULL) printf("Semaphore Create Fail!\n");
    else printf("Semaphore Create Success!\n");

    hHandle1 = OpenSemaphore(SYNCHRONIZE|SEMAPHORE_MODIFY_STATE,
               NULL,
               "SemaphoreName1");                    //打开信号量
    if   (hHandle1 == NULL)   printf("Semaphore Open Fail!\n");
    else printf("Semaphore Open Success!\n");

    h1 = CreateThread((LPSECURITY_ATTRIBUTES)NULL,
                      0,
                      (LPTHREAD_START_ROUTINE)func,
                      (LPVOID)NULL,
                      0,&dwThreadID1);                //创建子线程
    if (h1 == NULL) printf("Thread1 create Fail!\n");
    else printf("Thread1 create Success!\n");

    dRes = WaitForSingleObject(hHandle1,INFINITE);          //主线程等待子线程结束
    err = GetLastError();
    printf("WaitForSingleObject err = %d\n",err);

    if (dRes == WAIT_TIMEOUT)  printf("TIMEOUT!dRes = %d\n",dRes);
    else if  (dRes == WAIT_OBJECT_0) printf("WAIT_OBJECT!dRes = %d\n",dRes);
    else if  (dRes == WAIT_ABANDONED)
        printf("WAIT_ABANDONED!dRes = %d\n",dRes);
    else  printf("dRes = %d\n",dRes);

    CloseHandle(h1);
    CloseHandle(hHandle1);
    ExitThread(0);

    return nRetCode;
}

void func()
```

```
{
    BOOL rc;
    DWORD err;

    printf("Now In Thread!\n");
    rc = ReleaseSemaphore(hHandle1,1,NULL);                    //子线程唤醒主线程
    err = GetLastError();
    printf("ReleaseSemaphore err = %d\n",err);
    if (rc == 0) printf("Semaphore Release Fail!\n");
    else printf("Semaphore Release Success!rc = %d\n",rc);
}
```

2.2.8 实验展望

上面的程序完成了主、子两个线程执行先后顺序的同步关系，思考以下问题。

(1) 如何实现多个线程的同步？

(2) 若允许子线程执行多次后主线程再执行，又如何设置信号量的初值？

2.3 实验三：线程的互斥

2.3.1 实验目的

(1) 熟练掌握 Windows 系统环境下线程的创建与撤销。

(2) 熟悉 Windows 系统提供的线程互斥 API。

(3) 使用 Windows 系统提供的线程互斥 API 解决实际问题。

2.3.2 实验准备知识：相关 API 函数介绍

2.3.2.1 临界区对象

临界区对象(CriticalSection)包括初始化临界区(InitializeCriticalSection())、进入临界区(EnterCriticalSection())、退出临界区(LeaveCriticalSection())及删除临界区DeleteCriticalSection())等 API 函数。

1. 初始化临界区

InitializeCriticalSection()用于初始化临界区对象。

原型：

```
VOID InitializeCriticalSection(
 LPCRITICAL_SECTION lpCriticalSection                    // 指向临界区对象的地址指针
);
```

参数说明：

lpCriticalSection：指出临界区对象的地址。

返回值：

该函数没有返回值。

用法举例：

```
LPCRITICAL_SECTION hCriticalSection;                    //定义指向临界区对象的地址指针
CRITICAL_SECTION Critical;                              //定义一个临界区
hCriticalSection = &Critical;
InitializeCriticalSection(hCriticalSection);
```

2. 进入临界区

EnterCriticalSection()等待进入临界区的权限，当获得该权限后进入临界区。

原型：

```
VOID EnterCriticalSection(
 LPCRITICAL_SECTION lpCriticalSection                   // 指向临界区对象的地址指针
);
```

参数说明：

lpCriticalSection：指出临界区对象的地址。

返回值：

该函数没有返回值。

用法举例：

```
LPCRITICAL_SECTION hCriticalSection;                    //定义指向临界区对象的地址指针
CRITICAL_SECTION Critical;                              //定义一个临界区
hCriticalSection = &Critical;
EnterCriticalSection(hCriticalSection);
```

3. 退出临界区

LeaveCriticalSection()释放临界区的使用权限。

原型：

```
VOID LeaveCriticalSection(
 LPCRITICAL_SECTION lpCriticalSection                   // 指向临界区对象的地址指针
);
```

参数说明：

lpCriticalSection：指出临界区对象的地址。

返回值：

该函数没有返回值。

用法举例：

```
LPCRITICAL_SECTION hCriticalSection;                    //定义指向临界区对象的地址指针
CRITICAL_SECTION Critical;                              //定义一个临界区
hCriticalSection = &Critical;
LeaveCriticalSection(hCriticalSection);
```

4. 删除临界区

DeleteCriticalSection() 删除与临界区有关的所有系统资源。

原型：

```
VOID DeleteCriticalSection(
```

```
    LPCRITICAL_SECTION lpCriticalSection                    // 指向临界区对象的地址指针
);
```

参数说明：

lpCriticalSection：指出临界区对象的地址。

返回值：

该函数没有返回值。

用法举例：

```
LPCRITICAL_SECTION hCriticalSection;                        //定义指向临界区对象的地址指针
CRITICAL_SECTION Critical;                                  //定义一个临界区
hCriticalSection = &Critical;
DeleteCriticalSection(hCriticalSection);
```

2.3.2.2 互斥对象(Mutex)

互斥对象(Mutex)包括创建互斥对象(CreateMutex())、打开互斥对象(OpenMutex())及释放互斥对象(ReleaseMutex())API 函数。

1. 创建互斥对象

CreateMutex()用于创建一个互斥对象。

原型：

```
HANDLE CreateMutex(
 LPSECURITY_ATTRIBUTES lpMutexAttributes,                   // 安全属性
 BOOL bInitialOwner,                                        // 初始权限标志
 LPCTSTR lpName                                             // 互斥对象名
);
```

参数说明：

(1) lpMutexAttributes：指定安全属性，为 NULL 时，信号量得到一个默认的安全描述符。

(2) bInitialOwner：指定初始的互斥对象。如果该值为 TRUE 并且互斥对象已经存在，则调用线程获得互斥对象的所有权，否则调用线程不能获得互斥对象的所有权。想要知道互斥对象是否已经存在，参见返回值说明。

(3) lpName：给出互斥对象的名字。

返回值：

互斥对象创建成功，将返回该互斥对象的句柄。如果给出的互斥对象是系统已经存在的互斥对象，将返回这个已存在互斥对象的句柄。如果失败，系统返回 NULL，可以调用函数 GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;                             //定义一个句柄
// 创建一个名为"MutexName1"的互斥对象
hHandle1 = CreateMutex(NULL,FALSE, "MutexName1");
```

2. 打开互斥对象

OpenMutex()用于打开一个互斥对象。

原型：

```
HANDLE OpenMutex(
 DWORD dwDesiredAccess,                    // 访问标志
 BOOL bInheritHandle,                      // 继承标志
 LPCTSTR lpName                            // 互斥对象名
);
```

参数说明：

指明系统安全属性支持的对互斥对象所有可能的访问。如果系统安全属性不支持，则不能获得对互斥对象访问权。

(1) dwDesiredAccess：指出打开后要对互斥对象进行何种访问，具体描述如表 2-4 所示。

表 2-4 对互斥对象进行访问的种类

访　问	描　述
MUTEX_ALL_ACCESS	可以进行任何对互斥对象的访问
SYNCHRONIZE	使用等待函数 wait functions 等待互斥对象成为可用状态或使用 ReleaseMutex()释放使用权，从而获得互斥对象的使用权

(2) bInheritHandle：指出返回的信号量句柄是否可以继承。

(3) lpName：给出信号量的名字。

返回值：

互斥对象打开成功，将返回该互斥对象的句柄；如果失败，系统返回 NULL，可以调用函数 GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;
// 打开一个名为"MutexName1"的互斥对象
hHandle1 = OpenMutex(SYNCHRONIZE,NULL,"MutexName1");
```

3. 释放互斥对象

ReleaseMutex()用于释放互斥对象。

原型：

```
BOOL ReleaseMutex(
 HANDLE hMutex                             // 互斥对象句柄
);
```

参数说明：

hMutex：Mutex 对象的句柄。CreateMutex()和 OpenMutex()函数返回该句柄。

返回值：

如果成功，将返回一个非 0 值；如果失败，系统返回 0，可以调用函数 GetLastError()查询失败的原因。

用法举例：

```
static  HANDLE hHandle1 = NULL;
```

```
BOOL rc;
rc = ReleaseMutex(hHandle1)
```


2.3.3 实验内容

完成两个子线程之间的互斥。在主线程中使用系统调用 CreateThread()创建两个子线程，并使两个子线程互斥地使用全局变量 count。

2.3.4 实验要求

能正确使用临界区对象，包括初始化临界区 InitializeCriticalSection()、进入临界区 EnterCriticalSection()、退出临界区 LeaveCriticalSection()及删除临界区 DeleteCriticalSection()，进一步理解线程的互斥。

2.3.5 实验指导

具体操作过程同实验一，在 Microsoft Visual C++ 6.0 环境下建立一个 MFC 支持的控制台工程文件，编写 C 程序，在主线程中使用 InitializeCriticalSection()初始化临界区，然后建立两个子线程，在两个子线程中使用全局变量 count 的前、后分别使用 EnterCriticalSection()进入临界区及使用 LeaveCriticalSection()退出临界区，等两个子线程运行完毕，主线程使用 DeleteCriticalSection ()删除临界区并撤销线程。

2.3.6 实验总结

该实验完成了两个子线程的互斥。若去掉互斥对象，观察全局变量 count 的变化，了解互斥对象的作用，进一步理解线程的互斥。本实验也可以使用互斥对象(Mutex)来完成两个线程的互斥，互斥对象(Mutex)的使用方法与信号量对象相似，这里不再说明，请同学们自己完成。线程互斥访问全局变量 count 如图 2-5 所示。

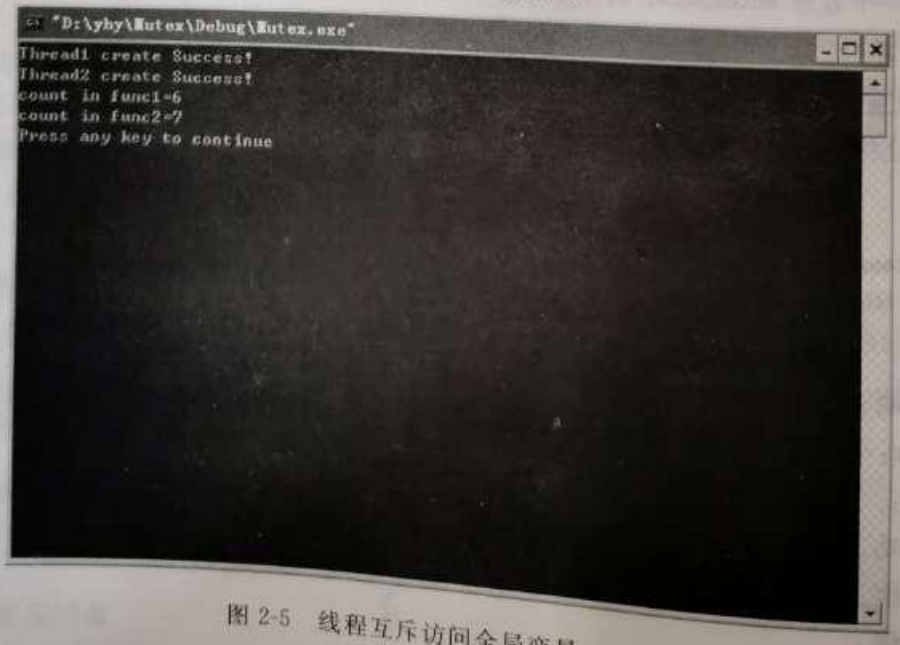

图 2-5 线程互斥访问全局变量 count

2.3.7 源程序

```
// Mutex.cpp: Defines the entry point for the console application.
//
#include "stdafx.h"
#include "Mutex.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////////
// The one and only application object
CWinApp theApp;
using namespace std;

static int count = 5;
static HANDLE h1;
static HANDLE h2;
LPCRITICAL_SECTION hCriticalSection;                    //定义指向临界区对象的地址指针
CRITICAL_SECTION Critical;                              //定义临界区
void func1();
void func2();

int _tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int nRetCode = 0;
    DWORD  dwThreadID1,dwThreadID2;
    hCriticalSection = &Critical;                       //将指向临界区对象的指针指向临界区
    InitializeCriticalSection(hCriticalSection);        //初始化临界区

    h1 = CreateThread((LPSECURITY_ATTRIBUTES)NULL,
                      0,
                      (LPTHREAD_START_ROUTINE)func1,
                      (LPVOID)NULL,
                      0,&dwThreadID1);                  //创建线程 func1
    if (h1 == NULL) printf("Thread1 create Fail!\n");
    else printf("Thread1 create Success!\n");

    h2 = CreateThread((LPSECURITY_ATTRIBUTES)NULL,
                      0,
                      (LPTHREAD_START_ROUTINE)func2,
                      (LPVOID)NULL,
                      0,&dwThreadID2);                  //创建线程 func2
    if (h2 == NULL) printf("Thread2 create Fail!\n");
    else printf("Thread2 create Success!\n");

    Sleep(1000);
```

```
    CloseHandle(h1);
    CloseHandle(h2);                                    //删除临界区
    DeleteCriticalSection(hCriticalSection);
    ExitThread(0);
    return nRetCode;
}

void func2()
{
    int r2;

    EnterCriticalSection(hCriticalSection);             //进入临界区
    r2 = count;
    _sleep(100);
    r2 = r2 + 1;
    count = r2;
    printf("count in func2 = %d\n",count);
    LeaveCriticalSection(hCriticalSection);             //退出临界区
}
void func1()
{
    int r1;

    EnterCriticalSection(hCriticalSection);             //进入临界区
    r1 = count;
    _sleep(500);
    r1 = r1 + 1;
    count = r1;
    printf("count in func1 = %d\n",count);
    LeaveCriticalSection(hCriticalSection);             //退出临界区
}
```

2.3.8 实验展望

上面的实验是使用临界区对象(CriticalSection)实现的,同学们可以用互斥对象(Mutex)来完成。

在完成以上3个实验后,同学们对Windows系统提供的线程的创建与撤销、线程的同步与互斥API有了一定的了解,在此基础上设计并完成一个综合性的实验,解决实际的同步与互斥问题,如生产者与消费者问题、读者与写者问题等。实验的题目可自行设计,但要求必须涉及线程的创建与撤销、等待对象函数(wait functions)、信号量对象(Semaphore)、临界区对象(CriticalSection)或互斥对象(Mutex)的使用。

2.4 实验四:使用命名管道实现进程通信

2.4.1 实验目的

(1) 了解Windows系统环境下的进程通信机制。

(2) 熟悉Windows系统提供的进程通信API。

2.4.2 实验准备知识：相关 API 函数介绍

1. 建立命名管道

函数 CreateNamePipe()创建一个命名管道实例，并返回该管道的句柄。

原型：

```
HANDLE CreateNamePipe(
 LPCTSTR lpName,                          // 命名管道的名字
 DWORD dwOpenMode,                        // 命名管道的访问模式
 DWORD dwPipeMode,                        // 命名管道的模式
 DWORD nMaxInstances,                     // 可创建实例的最大值
 DWORD nOutBufferSize,                    // 以字节为单位的输出缓冲区的大小
 DWORD nInBufferSize,                     // 以字节为单位的输入缓冲区的大小
 DWORD nDefaultTimeOut,                   // 默认超时时间
 LPSECURITY_ATTRIBUTES lpSecurityAttributes   // 安全属性
);
```

参数说明：

(1) lpName：为命名管道的名字，管道的命名方式为\\ServerName\pipe\pipename，其中 ServerName 为用命名管道通信时服务器的主机名或 IP 地址，pipename 为命名管道的名字，用户可自行定义。

(2) dwOpenMode：指出命名管道的访问模式。模式如表 2-5 所示。

表 2-5 管道的访问模式

模　式	说　明
PIPE_ACCESS_DUPLEX	双向管道。服务器和客户都可以进行读和写
PIPE_ACCESS_INBOUND	管道中数据的流向只能是从客户到服务器。服务器用 GENERIC_READ 模式创建管道，客户在连接管道时用 GENERIC_WRITE 模式
PIPE_ACCESS_OUTBOUND	管道中数据的流向只能是从服务器到客户。服务器用 GENERIC_WRITE 模式创建管道，客户在连接管道时用 GENERIC_READ 模式

(3) dwPipeMode：指出管道的模式。模式如表 2-6 所示。

表 2-6 管道的模式

模　式	说　明
PIPE_TYPE_BYTE	以字符流的方式向管道写数据。该模式下不能使用 PIPE_READMODE_MESSAGE
PIPE_TYPE_MESSAGE	以消息流的方式向管道写数据。该模式下可以使用 PIPE_READMODE_MESSAGE 和 PIPE_READMODE_BYTE

(4) nMaxInstances：该命名管道可以创建实例的最大值。

(5) nOutBufferSize：输出缓冲区的大小，以字节为单位。

(6) nInBufferSize：输入缓冲区的大小，以字节为单位。

(7) nDefaultTimeOut：默认的超时时间，以 ms 为单位。如果函数 WaitNamedPipe() 指出 NMPWAIT_USE_DEFAULT_WAIT，每个管道实例必须指定同一值的名字。

(8) lpSecurityAttributes：为管道指定安全属性，为 NULL 时，管道得到一个默认的安全描述符。

返回值：

如果管道创建成功，将返回服务器命名管道实例的句柄。如果失败，返回 INVALID_HANDLE_VALUE，可以调用函数 GetLastError() 查询失败的原因；当返回 ERROR_INVALID_PARAMETER 时，表明参数 nMaxInstances 指定的值大于 PIPE_UNLIMITED_INSTANCES。

2. 连接命名管道

服务器用函数 ConnectNamePipe() 连接命名管道。创建后的命名管道也等待客户端的连接，客户端可以使用函数 CreateFile() 和 CallNamedPipe() 进行连接。

原型：

```
BOOL ConnectNamedPipe(
 HANDLE hNamedPipe,                          // 命名管道实例句柄
 LPOVERLAPPED lpOverlapped                   // 指向 Overlapped 结构的指针
);
```

参数说明：

(1) hNamedPipe：为命名管道创建时得到的一个命名管道实例句柄。

(2) lpOverlapped：指向 Overlapped 结构的指针，可设其为 NULL。

返回值：

成功，将返回一个非 0 值；失败，系统返回 0，可以调用函数 GetLastError() 查询失败的原因。

3. 拆除命名管道的连接

函数 DisconnectNamePipe() 拆除命名管道服务器与客户端的连接。

原型：

```
BOOL DisconnectNamedPipe(
 HANDLE hNamedPipe                           // 命名管道实例句柄
);
```

参数说明：

hNamedPipe：为命名管道创建时得到的一个命名管道实例句柄。

返回值：

成功，将返回一个非 0 值；失败，系统返回 0，可以调用函数 GetLastError() 查询失败的原因。

4. 客户端连接服务器已建立的命名管道

客户端使用函数 CallNamePipe() 连接服务器建立的命名管道。

原型：

```
BOOL CallNamedPipe(
 LPCTSTR lpNamedPipeName,                    // 命名管道的名字
 LPVOID lpInBuffer,                          // 输出数据缓冲区指针
```

(7) nDefaultTimeOut：默认的超时时间，以 ms 为单位。如果函数 WaitNamedPipe() 指出 NMPWAIT_USE_DEFAULT_WAIT，每个管道实例必须指定同一值的名字。

(8) lpSecurityAttributes：为管道指定安全属性，为 NULL 时，管道得到一个默认的安全描述符。

返回值：

如果管道创建成功，将返回服务器命名管道实例的句柄。如果失败，返回 INVALID_HANDLE_VALUE，可以调用函数 GetLastError() 查询失败的原因；当返回 ERROR_INVALID_PARAMETER 时，表明参数 nMaxInstances 指定的值大于 PIPE_UNLIMITED_INSTANCES。

2. 连接命名管道

服务器用函数 ConnectNamePipe() 连接命名管道。创建后的命名管道也等待客户端的连接，客户端可以使用函数 CreateFile() 和 CallNamedPipe() 进行连接。

原型：

```
BOOL ConnectNamedPipe(
 HANDLE hNamedPipe,                          // 命名管道实例句柄
 LPOVERLAPPED lpOverlapped                   // 指向 Overlapped 结构的指针
);
```

参数说明：

(1) hNamedPipe：为命名管道创建时得到的一个命名管道实例句柄。

(2) lpOverlapped：指向 Overlapped 结构的指针，可设其为 NULL。

返回值：

成功，将返回一个非 0 值；失败，系统返回 0，可以调用函数 GetLastError() 查询失败的原因。

3. 拆除命名管道的连接

函数 DisconnectNamePipe() 拆除命名管道服务器与客户端的连接。

原型：

```
BOOL DisconnectNamedPipe(
 HANDLE hNamedPipe                           // 命名管道实例句柄
);
```

参数说明：

hNamedPipe：为命名管道创建时得到的一个命名管道实例句柄。

返回值：

成功，将返回一个非 0 值；失败，系统返回 0，可以调用函数 GetLastError() 查询失败的原因。

4. 客户端连接服务器已建立的命名管道

客户端使用函数 CallNamePipe() 连接服务器建立的命名管道。

原型：

```
BOOL CallNamedPipe(
 LPCTSTR lpNamedPipeName,
 LPVOID lpInBuffer,                          // 命名管道的名字
                                             // 输出数据缓冲区指针
```

```
    DWORD nInBufferSize,                    // 以字节单位的输出数据缓冲区的大小
    LPVOID lpOutBuffer,                     // 输入数据缓冲区指针
    DWORD nOutBufferSize,                   // 以字节单位的输入数据缓冲区的大小
    LPDWORD lpBytesRead,                    // 输入字节数指针
    DWORD nTimeOut                          // 等待时间
);
```

参数说明：

(1) lpNamedPipeName：命名管道的名字。

(2) lpInBuffer：指出用于输出数据(向管道写数据)的缓冲区指针。

(3) nInBufferSize：用于输出数据缓冲区的大小，以字节单位。

(4) lpOutBuffer：指出用于接收数据(从管道读出数据)的缓冲区指针。

(5) nOutBufferSize：指向用于接收数据缓冲区的大小，以字节单位。

(6) lpBytesRead：一个 32 位的变量，该变量用于存储从管道读出的字节数。

(7) nTimeOut：等待命名管道成为可用状态的时间，单位为 ms。

返回值：

成功，将返回一个非 0 值；失败，系统返回 0，可以调用函数 GetLastError()查询失败的原因。

5. 客户端等待命令管道

客户端使用函数 WaitNamedPipe()等待服务器连接命名管道。

原型：

```
BOOL WaitNamedPipe(
  LPCTSTR lpNamedPipeName,                  // 要等待的命名管道名
  DWORD nTimeOut                            // 等待时间
);
```

参数说明：

(1) lpNamedPipeName：要等待的命名管道的名字。

(2) nTimeOut：等待命名管道成为可用状态的时间，单位为 ms。

返回值：

在等待时间内要连接的命名管道可以使用，将返回一个非 0 值。在等待时间内要连接的命名管道不可以使用，系统返回 0，可以调用函数 GetLastError()查询失败的原因。

2.4.3 实验内容

使用命名管道完成两个进程之间的通信。

2.4.4 实验要求

使用 Windows 系统提供的命名管道完成两个进程之间的通信，要求能正确使用创建命名管道 CreateNamePipe()、连接命名管道 ConnectNamePipe()、拆除命名管道的连接 DisconnectNamePipe()、连接服务器已建立的命名管道 CallNamePipe()、等待命名管道 WaitNamedPipe()等 API。

2.4.5 实验指导

完成两个进程之间的通信，需要建立两个工程文件，在 Microsoft Visual C++ 6.0 环境

下建立服务器工程文件 PipeServer 和客户端工程文件 PipeClient。在服务器程序中，首先使用 CreateNamePipe()创建一个命名管道，之后使用 ConnectNamePipe()连接命名管道，如果命名管道连接成功，可以使用读文件函数 ReadFile()从管道中读取数据，并可使用写文件函数 WriteFile()向管道中写入数据，管道使用完毕后可以使用 DisconnectNamePipe()拆除与命名管道的连接。在客户端程序中，可以先使用 WaitNamedPipe()等待服务器建立好命名管道，然后使用 CallNamePipe()与服务器建立命名管道的连接，并同时得到服务器发来的数据或向服务器发送数据，如图 2-6 所示。

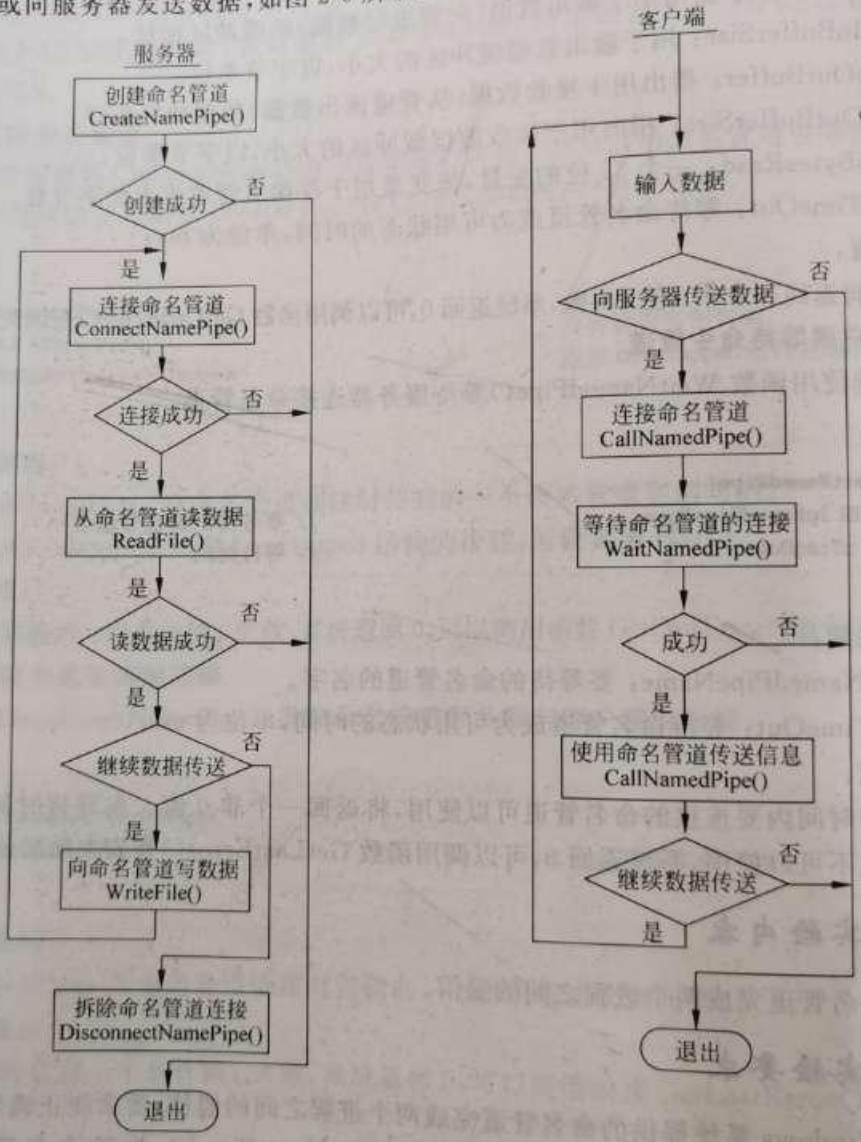

图 2-6　进程通信流程图

2.4.6　实验总结

该实验完成了两个进程的通信，请同学们在下面程序的基础上增加和完善程序的功能，

如设计一个聊天室等，使其可以实现自己的设计需求。命名管道创建命令 CreateNamePipe() 也可以使用文件创建命令 CreateFile()来实现其功能，同学们可以自己去尝试，命名管道创建命令 CreateNamePipe()中的参数比较多，请同学们仔细研究其含义，使用不当可能会导致两个进程通信的失败。

图 2-7 所示为客户端程序运行情况，首先在客户端输入数据 HelloServer!，按回车键后结果见图 2-8，可以看到客户端输入的数据已经在服务器端显示出来了。同样在服务器输入数据 HelloClient!，在客户端同样也显示出来，这说明建立的命名管道已经做到了双向通信。

图 2-7 命名管道客户端运行情况

图 2-8 命名管道服务器运行情况

2.4.7 源程序

```
/********** 服务器程序 **********/
// PipeServer.cpp: Defines the entry point for the console application.
//
#include "stdafx.h"
#include "PipeServer.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////////////
// The one and only application object
CWinApp theApp;
using namespace std;

int _tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int nRetCode = 0;
    int err;
    BOOL rc;
    HANDLE hPipeHandle1;

    char lpName[] = "\\\\.\\pipe\\myPipe";
    char InBuffer[50] = "";
    char OutBuffer[50] = "";
    DWORD BytesRead,BytesWrite;

    hPipeHandle1 = CreateNamedPipe(
        (LPCTSTR)lpName,
        PIPE_ACCESS_DUPLEX|FILE_FLAG_OVERLAPPED|WRITE_DAC,
        PIPE_TYPE_MESSAGE|PIPE_READMODE_BYTE|PIPE_WAIT,
        1,20,30, NMPWAIT_USE_DEFAULT_WAIT,
        (LPSECURITY_ATTRIBUTES)NULL
        );
                                                    //创建命名管道
    if ((hPipeHandle1 == INVALID_HANDLE_VALUE) || (hPipeHandle1 == NULL))
    {    err = GetLastError();
         printf("Server Pipe Create Fail!err = %d\n",err);
         exit(1);
    }
    else printf("Server Pipe Create Success!\n");

    while (1)
    {   //连接命名管道
        rc = ConnectNamedPipe(hPipeHandle1,(LPOVERLAPPED)NULL);
        if (rc == 0)
        {
```

```
            err = GetLastError();
            printf("Server Pipe Connect Fail err = %d\n",err);
            exit(2);
        }
        else printf("Server Pipe Connect Success\n");
        strcpy(InBuffer,"");
        strcpy(OutBuffer,"");
        //从命名管道读数据
        rc = ReadFile(hPipeHandle1,InBuffer,sizeof(InBuffer),&BytesRead,
                (LPOVERLAPPED)NULL);
        if (rc == 0 && BytesRead == 0)
        {   err = GetLastError();
            printf("Server Read Pipe Fail!err = %d\n",err);
            exit(3);
        }
        else
        {
              printf("Server Read Pipe Success!\nDATA from Client is = %s\n",InBuffer);
        }
        rc = strcmp(InBuffer,"end");
        if (rc == 0)  break;
        printf("Please Input Data to Send\n");
        scanf("%s",OutBuffer);
        //向命名管道写数据
        rc = WriteFile(hPipeHandle1,OutBuffer,sizeof(OutBuffer),&BytesWrite,
              (LPOVERLAPPED)NULL);
        if (rc == 0) printf("Server Write Pipe Fail!\n");
        else printf("Server Write Pipe Success!\n");
        DisconnectNamedPipe(hPipeHandle1);          //拆除与命名管道的连接
        rc = strcmp(OutBuffer,"end");
        if (rc == 0)  break;

    }
    printf("Now Server be END!\n");
    CloseHandle(hPipeHandle1);
    return nRetCode;
}

/********** 客户端程序 **********/
// PipeClient.cpp: Defines the entry point for the console application.
//
#include "stdafx.h"
#include "PipeClient.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
// The one and only application object
CWinApp theApp;
using namespace std;

int _tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    BOOL rc = 0;
    char lpName[] = "\\\\.\\pipe\\myPipe";
    char InBuffer[50] = "";
    char OutBuffer[50] = "";
    DWORD BytesRead;

    int nRetCode = 0;    int err = 0;
    while (1)
    {
        strcpy(InBuffer,"");
        strcpy(OutBuffer,"");
        printf("Input Data Please!\n");
        scanf("%s",InBuffer);
        rc = strcmp(InBuffer,"end");
        if  (rc == 0)
        {   //连接命名管道
            rc = CallNamedPipe(lpName,InBuffer, sizeof(InBuffer),OutBuffer,
                sizeof(OutBuffer), &BytesRead, NMPWAIT_USE_DEFAULT_WAIT);
            break;
        }
        rc = WaitNamedPipe(lpName,NMPWAIT_WAIT_FOREVER);          //等待命名管道
        if (rc == 0)
        {
            err = GetLastError();
            printf("Wait Pipe Fail!err = %d\n",err);
            exit(1);
        }
        else printf("Wait Pipe Success!\n");
        //使用命名管道读/写数据
        rc = CallNamedPipe(lpName,InBuffer, sizeof(InBuffer),OutBuffer,
              sizeof(OutBuffer), &BytesRead, NMPWAIT_USE_DEFAULT_WAIT);
        rc = strcmp(OutBuffer,"end");
        if  (rc == 0)  break;
        if (rc == 0)
        {
            err = GetLastError();
            printf("Pipe Call Fail!err = %d\n",err);
            exit(1);
        }
        else printf("Pipe Call Success!\nData from Server is %s\n",OutBuffer);
    }
    printf("Now Client to be End!\n");
    return nRetCode;
}
```