

深入浅出 Win32 多线程程序设计之基本概念

引言

从单进程单线程到多进程多线程是操作系统发展的一种必然趋势,当年的 DOS 系统属于单任务操作系统,最优秀的程序员也只能通过驻留内存的方式实现所谓的"多任务",而如今的 Win32 操作系统却可以一边听音乐,一边编程,一边打印文档。

理解多线程及其同步、互斥等通信方式是理解现代操作系统的关键一环,当我们精通了 Win32 多线程程序设计后,理解和学习其它操作系统的多任务控制也非常容易。许多程序员从来没有学习过嵌入式系统领域著名的操作系统 VxWorks,但是立马就能在上面做开发,大概要归功于平时在 Win32 多线程上下的功夫。

因此,学习 Win32 多线程不仅对理解 Win32 本身有重要意义,而且对学习和领会其它操作系统也有触类旁通的作用。

进程与线程

先阐述一下进程和线程的概念和区别,这是一个许多大学老师也讲不清楚的问题。

进程 (Process) 是具有一定独立功能的程序关于某个数据集合上的一次运行活动,是系统进行资源分配和调度的一个独立单位。程序只是一组指令的有序集合,它本身没有任何运行的含义,只是一个静态实体。而进程则不同,它是程序在某个数据集上的执行,是一个动态实体。它因创建而产生,因调度而运行,因等待资源或事件而被处于等待状态,因完成任务而被撤消,反映了一个程序在一定的数据集上运行的全部动态过程。

线程 (Thread) 是进程的一个实体,是 CPU 调度和分派的基本单位。线程不能够独立执行,必须依存在应用程序中,由应用程序提供多个线程执行控制。

线程和进程的关系是:线程是属于进程的,线程运行在进程空间内,同一进程所产生的线程共享同一内存空间,当进程退出时该进程所产生的线程都会被强制退出并清除。线程可与属于同一进程的其它线程共享进程所拥有的全部资源,但是其本身基本上不拥有系统资源,只拥有一点在运行中必不可少的信息(如程序计数器、一组寄存器和栈)。

根据进程与线程的设置,操作系统大致分为如下类型:

- (1) 单进程、单线程, MS-DOS 大致是这种操作系统;
- (2) 多进程、单线程, 多数 UNIX (及类 UNIX 的 LINUX) 是这种操作系统;
- (3) 多进程、多线程, Win32 (Windows NT/2000/XP 等)、Solaris 2.x 和 OS/2 都是这种操作系统;
- (4) 单进程、多线程, VxWorks 是这种操作系统。

在操作系统中引入线程带来的主要好处是:

- (1) 在进程内创建、终止线程比创建、终止进程要快;
- (2) 同一进程内的线程间切换比进程间的切换要快,尤其是用户级线程间的切换。另外,线程的出现还因为以下几个原因:

(1) 并发程序的并发执行,在多处理环境下更为有效。一个并发程序可以建立一个进程,而这个并发程序中的若干并发程序段就可以分别建立若干线程,使这些线程在不同的处理机上执行。

(2) 每个进程具有独立的地址空间,而该进程内的所有线程共享该地址空间。这样可以解决父子进程模型中,子进程必须复制父进程地址空间的问题。

- (3) 线程对解决客户/服务器模型非常有效。

Win32 进程

1、进程间通信（IPC）

Win32 进程间通信的方式主要有：

- (1) 剪贴板(Clip Board);
- (2) 动态数据交换(Dynamic Data Exchange);
- (3) 部件对象模型(Component Object Model);
- (4) 文件映射(File Mapping);
- (5) 邮件槽(Mail Slots);
- (6) 管道(Pipes);
- (7) Win32 套接字(Socket);
- (8) 远程过程调用(Remote Procedure Call);
- (9) WM_COPYDATA 消息(WM_COPYDATA Message)。

2、获取进程信息

在 WIN32 中，可使用在 PSAPI .DLL 中提供的 Process status Helper 函数帮助我们获取进程信息。

- (1) EnumProcesses()函数可以获取进程的 ID，其原型为：

```
BOOL EnumProcesses(DWORD * lpidProcess, DWORD cb, DWORD*cbNeeded);
```

参数 lpidProcess：一个足够大的 DWORD 类型的数组，用于存放进程的 ID 值；

参数 cb：存放进程 ID 值的数组的最大长度，是一个 DWORD 类型的数据；

参数 cbNeeded：指向一个 DWORD 类型数据的指针，用于返回进程的数目；

函数返回值：如果调用成功，返回 TRUE，同时将所有进程的 ID 值存放在 lpidProcess 参数所指向的数组中，进程个数存放在 cbNeeded 参数所指向的变量中；如果调用失败，返回 FALSE。

- (2) GetModuleFileNameExA()函数可以实现通过进程句柄获取进程文件名，其原型为：

```
DWORD GetModuleFileNameExA(HANDLE hProcess, HMODULE hModule,LPTSTR  
lpstrFileName, DWORD nsize);
```

参数 hProcess：接受进程句柄的参数，是 HANDLE 类型的变量；

参数 hModule：指针型参数，在本文的程序中取值为 NULL；

参数 lpstrFileName：LPTSTR 类型的指针，用于接受主调函数传递来的用于存放进程名的字符数组指针；

参数 nsize：lpstrFileName 所指数组的长度；

函数返回值：如果调用成功，返回一个大于 0 的 DWORD 类型的数据，同时将 hProcess 所对应的进程名存放在 lpstrFileName 参数所指向的数组中；如果调用失败，则返回 0。

通过下列代码就可以遍历系统中的进程，获得进程列表：

```
//获取当前进程总数  
EnumProcesses(process_ids, sizeof(process_ids), &num_processes);
```

```

//遍历进程
for (int i = 0; i < num_processes; i++)
{
    //根据进程 ID 获取句柄
    process[i] = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, 0,
                             process_ids[i]);
    //通过句柄获取进程文件名
    if (GetModuleFileNameExA(process[i], NULL, File_name, sizeof(fileName)))
        cout << fileName << endl;
}

```

Win32 线程

WIN32 靠线程的优先级（达到抢占式多任务的目的）及分配给线程的 CPU 时间来调度线程。WIN32 本身的许多应用程序也利用了多线程的特性，如任务管理等。

本质而言，一个处理器同一时刻只能执行一个线程（“微观串行”）。WIN32 多任务机制使得 CPU 好像在同时处理多个任务一样，实现了“宏观并行”。其多线程调度的机制为：

- （1）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；
- （2）保存当前执行线程的描述表(上下文)；
- （3）装入下一执行线程的描述表(上下文)；
- （4）若存在等待被执行的线程，则重复上述过程。

WIN32 下的线程可能具有不同的优先级，优先级的范围为 0~31，共 32 级，其中 31 表示最高优先级，优先级 0 为系统保留。它们可以分成两类，即实时优先级和可变优先级：

- （1）实时优先级从 16 到 31，是实时程序所用的高优先级线程，如许多监控类应用程序；
- （2）可变优先级从 1 到 15，绝大多数程序的优先级都在这个范围内。。WIN32 调度器为了优化系统响应时间，在它们执行过程中可动态调整它们的优先级。

多线程确实给应用开发带来了许多好处，但并非任何情况下都要使用多线程，一定要根据应用程序的具体情况来综合考虑。一般来说，在以下情况下可以考虑使用多线程：

- （1）应用程序中的各任务相对独立；
- （2）某些任务耗时较多；
- （3）各任务需要有不同的优先级。

另外，对于一些实时系统应用，应考虑多线程。

Win32 核心对象

WIN32 核心对象包括进程、线程、文件、事件、信号量、互斥体和管道，核心对象可能有不只一个拥有者，甚至可以跨进程。有一组 WIN32 API 与核心对象息息相关：

- （1）WaitForSingleObject，用于等待对象的“激活”，其函数原型为：

```

DWORD WaitForSingleObject(
    HANDLE hHandle, // 等待对象的句柄
    DWORD dwMilliseconds // 等待毫秒数，INFINITE 表示无限等待
);

```

可以作为 WaitForSingleObject 第一个参数的对象包括：Change notification、Console input、Event、Job、Memory resource notification、Mutex、Process、Semaphore、Thread 和 Waitable timer。

如果等待的对象不可用，那么线程就会挂起，直到对象可用线程才会被唤醒。对不同的对象，WaitForSingleObject 表现为不同的含义。例如，使用 WaitForSingleObject(hThread,...)可以判断一个线程是否结束；使用 WaitForSingleObject(hMutex,...)可以判断是否能够进入临界区；而 WaitForSingleObject(hProcess,...)则表现为等待一个进程的结束。

与 WaitForSingleObject 对应还有一个 WaitForMultipleObjects 函数，可以用于等待多个对象，其原型为：

```
DWORD WaitForMultipleObjects(DWORD nCount,const HANDLE* pHandles,BOOL
                               bWaitAll,DWORD dwMilliseconds);
```

(2) CloseHandle，用于关闭对象，其函数原型为：

```
BOOL CloseHandle(HANDLE hObject);
```

如果函数执行成功，则返回 TRUE；否则返回 FALSE，我们可以通过 GetLastError 函数进一步可以获得错误原因。

C 运行时库

在 VC++6.0 中，有两种多线程编程方法：一是使用 C 运行时库及 WIN32 API 函数，另一种方法是使用 MFC，MFC 对多线程开发有强大的支持。

标准 C 运行时库是 1970 年问世的，当时还没有多线程的概念。因此，C 运行时库早期的设计者们不可能考虑到让其支持多线程应用程序。

Visual C++提供了两种版本的 C 运行时库，一个版本供单线程应用程序调用，另一个版本供多线程应用程序调用。多线程运行时库与单线程运行时库有两个重大差别：

- (1) 类似 errno 的全局变量，每个线程单独设置一个；这样从每个线程中可以获取正确的错误信息。
- (2) 多线程库中的数据结构以同步机制加以保护。这样可以避免访问时候的冲突。

Visual C++提供的多线程运行时库又分为静态链接库和动态链接库两类，而每一类运行时库又可再分为 debug 版和 release 版，因此 Visual C++共提供了 6 个运行时库。如下表：

C 运行时库	库文件
Single thread(static link)	libc.lib
Debug single thread(static link)	Libcd.lib
MultiThread(static link)	libcmtd.lib
Debug multiThread(static link)	libcmtd.lib
MultiThread(dynamic link)	msvert.lib
Debug multiThread(dynamic link)	msvertd.lib

如果不使用 VC 多线程 C 运行时库来生成多线程程序，必须执行下列操作：

- (1) 使用标准 C 库（基于单线程）并且只允许可重入函数集进行库调用；
- (2) 使用 Win32 API 线程管理函数，如 CreateThread；
- (3) 通过使用 Win32 服务（如信号量和 EnterCriticalSection 及 LeaveCriticalSection 函数），为不可重入的函数提供自己的同步。

如果使用标准 C 库而调用 VC 运行时库函数，则在程序的 link 阶段会提示如下错误：

```
error LNK2001: unresolved external symbol __endthreadex
error LNK2001: unresolved external symbol __beginthreadex
```

深入浅出 Win32 多线程程序设计之线程控制

WIN32 线程控制主要实现线程的创建、终止、挂起和恢复等操作，这些操作都依赖于 WIN32 提供的一组 API 和具体编译器的 C 运行时库函数。

1. 线程函数

在启动一个线程之前，必须为线程编写一个全局的线程函数，这个线程函数接受一个 32 位的 LPVOID 作为参数，返回一个 UINT，线程函数的结构为：

```
UINT ThreadFunction(LPVOID pParam)
{
    //线程处理代码
    return 0;
}
```

在线程处理代码部分通常包括一个死循环，该循环中先等待某事情的发生，再处理相关的工作：

```
while(1)
{
    WaitForSingleObject(.....); //或 WaitForMultipleObjects(...)
    //Do something
}
```

一般来说，C++ 的类成员函数不能作为线程函数。这是因为在类中定义的成员函数，编译器会为其加上 this 指针。请看下列程序：

```
#include "windows.h"
#include <process.h>
class ExampleTask
{
public:
```

```

        void taskmain(LPVOID param);
        void StartTask();
    };
void ExampleTask::taskmain(LPVOID param)
{

}

void ExampleTask::StartTask()
{
    _beginthread(taskmain,0,NULL);
}

int main(int argc, char* argv[])
{
    ExampleTask realTimeTask;
    realTimeTask.StartTask();
    return 0;
}

```

程序编译时出现如下错误:

```

error C2664: '_beginthread' : cannot convert parameter 1 from 'void (void *)' to 'void (__cdecl *)(void
*)'
None of the functions with this name in scope match the target type

```

再看下列程序:

```

#include "windows.h"
#include <process.h>
class ExampleTask
{
public:
    void taskmain(LPVOID param);
};

void ExampleTask::taskmain(LPVOID param)
{

}

int main(int argc, char* argv[])
{
    ExampleTask realTimeTask;
    _beginthread(ExampleTask::taskmain,0,NULL);
    return 0;
}

```

程序编译时会出错：

```
error C2664: '_beginthread' : cannot convert parameter 1 from 'void (void *)' to 'void (__cdecl *)(void
*)'
None of the functions with this name in scope match the target type
```

如果一定要以类成员函数作为线程函数，通常有如下解决方案：

(1) 将该成员函数声明为 **static** 类型，去掉 **this** 指针；

我们将上述二个程序改变为：

```
#include "windows.h"
#include <process.h>
class ExampleTask
{
public:
void static taskmain(LPVOID param);
void StartTask();
};

void ExampleTask::taskmain(LPVOID param)
{
}

void ExampleTask::StartTask()
{
_beginthread(taskmain,0,NULL);
}

int main(int argc, char* argv[])
{
ExampleTask realTimeTask;
realTimeTask.StartTask();
return 0;
}
和
#include "windows.h"
#include <process.h>
class ExampleTask
{
public:
void static taskmain(LPVOID param);
};

void ExampleTask::taskmain(LPVOID param)
{
}
```

```

int main(int argc, char* argv[])
{
    _beginthread(ExampleTask::taskmain,0,NULL);
    return 0;
}

```

均编译通过。

将成员函数声明为静态虽然可以解决作为线程函数的问题，但是它带来了新的问题，那就是 `static` 成员函数只能访问 `static` 成员。解决此问题的一种途径是可以在调用类静态成员函数（线程函数）时将 `this` 指针作为参数传入，并在改线程函数中用强制类型转换将 `this` 转换成指向该类的指针，通过该指针访问非静态成员。

(2) 不定义类成员函数为线程函数，而将线程函数定义为类的友元函数。这样，线程函数也可以有类成员函数同等的权限；

我们将程序修改为：

```

#include "windows.h"
#include <process.h>
class ExampleTask
{
public:
friend void taskmain(LPVOID param);
void StartTask();
};

void taskmain(LPVOID param)
{
    ExampleTask * pTaskMain = (ExampleTask *) param;
    //通过 pTaskMain 指针引用
}

void ExampleTask::StartTask()
{
    _beginthread(taskmain,0,this);
}

int main(int argc, char* argv[])
{
    ExampleTask realTimeTask;
    realTimeTask.StartTask();
    return 0;
}

```

(3) 可以对非静态成员函数实现回调，并访问非静态成员，此法涉及到一些高级技巧，在此不再详述。

2. 创建线程

进程的主线程由操作系统自动生成，Win32 提供了 CreateThread API 来完成用户线程的创建，该 API 的原型为：

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, //Pointer to a SECURITY_ATTRIBUTES  
        structure  
    SIZE_T dwStackSize, //Initial size of the stack, in bytes.  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter, //Pointer to a variable to be passed to the thread  
    DWORD dwCreationFlags, //Flags that control the creation of the thread  
    LPDWORD lpThreadId //Pointer to a variable that receives the thread identifier  
);
```

如果使用 C/C++ 语言编写多线程应用程序，一定不能使用操作系统提供的 CreateThread API，而应该使用 C/C++ 运行时库中的 _beginthread（或 _beginthreadex），其函数原型为：

```
uintptr_t _beginthread(  
    void( __cdecl *start_address )( void * ), //Start address of routine that begins execution of new  
        thread  
    unsigned stack_size, //Stack size for new thread or 0.  
    void *arglist //Argument list to be passed to new thread or NULL  
);  
uintptr_t _beginthreadex(  
    void *security, //Pointer to a SECURITY_ATTRIBUTES structure  
    unsigned stack_size,  
    unsigned ( __stdcall *start_address )( void * ),  
    void *arglist,  
    unsigned initflag, //Initial state of new thread (0 for running or CREATE_SUSPENDED for  
        suspended);  
    unsigned *thrdaddr  
);
```

_beginthread 函数与 Win32 API 中的 CreateThread 函数类似，但有如下差异：

- （1）通过 _beginthread 函数我们可以利用其参数列表 arglist 将多个参数传递到线程；
- （2）_beginthread 函数初始化某些 C 运行时库变量，在线程中若需要使用 C 运行时库。

3. 终止线程

线程的终止有如下四种方式：

- （1）线程函数返回；
- （2）线程自身调用 ExitThread 函数即终止自己，其原型为：

```
VOID ExitThread(UINT fuExitCode );
```

它将参数 `fuExitCode` 设置为线程的退出码。

注意：如果使用 C/C++ 编写代码，我们应该使用 C/C++ 运行时库函数 `_endthread` (`_endthreadex`) 终止线程，决不能使用 `ExitThread`！

`_endthread` 函数对于线程内的条件终止很有用。例如，专门用于通信处理的线程若无法获取对通信端口的控制，则会退出。

(3) 同一进程或其他进程的线程调用 `TerminateThread` 函数，其原型为：

```
BOOL TerminateThread(HANDLE hThread,DWORD dwExitCode);
```

该函数用来结束由 `hThread` 参数指定的线程，并把 `dwExitCode` 设成该线程的退出码。当某个线程不再响应时，我们可以用其他线程调用该函数来终止这个不响应的线程。

(4) 包含线程的进程终止。

最好使用第 1 种方式终止线程，第 2~4 种方式都不宜采用。

4. 挂起与恢复线程

当我们创建线程的时候，如果给其传入 `CREATE_SUSPENDED` 标志，则该线程创建后被挂起，我们应使用 `ResumeThread` 恢复它：

```
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 `0x FFFFFFFF`。

对于没有被挂起的线程，程序员可以调用 `SuspendThread` 函数强行挂起之：

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

一个线程可以被挂起多次。线程可以自行暂停运行，但是不能自行恢复运行。如果一个线程被挂起 `n` 次，则该线程也必须被恢复 `n` 次才可能得以执行。

5. 设置线程优先级

当一个线程被首次创建时，它的优先级等同于它所属进程的优先级。在单个进程内可以通过调用 `SetThreadPriority` 函数改变线程的相对优先级。一个线程的优先级是相对于其所属进程的优先级而言的。

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

其中参数 `hThread` 是指向待修改优先级线程的句柄，线程与包含它的进程的优先级关系如下：

线程优先级 = 进程类基本优先级 + 线程相对优先级

进程类的基本优先级包括：

- (1) 实时： `REALTIME_PRIORITY_CLASS`;
- (2) 高： `HIGH_PRIORITY_CLASS`;
- (3) 高于正常： `ABOVE_NORMAL_PRIORITY_CLASS`;
- (4) 正常： `NORMAL_PRIORITY_CLASS`;

- (5) 低于正常: BELOW_NORMAL_PRIORITY_CLASS;
- (6) 空闲: IDLE_PRIORITY_CLASS。

我们从 Win32 任务管理器中可以直观的看到这六个进程类优先级，如下图：



线程的相对优先级包括：

- (1) 空闲: THREAD_PRIORITY_IDLE;
- (2) 最低线程: THREAD_PRIORITY_LOWEST;
- (3) 低于正常线程: THREAD_PRIORITY_BELOW_NORMAL;
- (4) 正常线程: THREAD_PRIORITY_NORMAL (缺省);
- (5) 高于正常线程: THREAD_PRIORITY_ABOVE_NORMAL;
- (6) 最高线程: THREAD_PRIORITY_HIGHEST;
- (7) 关键时间: THREAD_PRIORITY_CRITICAL。

下图给出了进程优先级和线程相对优先级的映射关系：

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

例如：

```
HANDLE hCurrentThread = GetCurrentThread();  
//获得该线程句柄  
SetThreadPriority(hCurrentThread, THREAD_PRIORITY_LOWEST);
```

6.睡眠

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

该函数可使线程暂停自己的运行，直到 dwMilliseconds 毫秒过去为止。它告诉系统，自身不想在某个时间段内被调度。

7.其它重要 API

获得线程优先级

一个线程被创建时，就会有一个默认的优先级，但是有时要动态地改变一个线程的优先级，有时需获得一个线程的优先级。

```
Int GetThreadPriority (HANDLE hThread);
```

如果函数执行发生错误，会返回 THREAD_PRIORITY_ERROR_RETURN 标志。如果函数成功地执行，会返回优先级标志。

获得线程退出码

```
BOOL WINAPI GetExitCodeThread(  
    HANDLE hThread,  
    LPDWORD lpExitCode  
);
```

如果执行成功，GetExitCodeThread 返回 TRUE，退出码被 lpExitCode 指向内存记录；否则返回 FALSE，我们可通过 GetLastError()获知错误原因。如果线程尚未结束，lpExitCode 带回来的将是 STILL_ALIVE。

```
获得/设置线程上下文  
BOOL WINAPI GetThreadContext(  
    HANDLE hThread,  
    LPCONTEXT lpContext  
);  
BOOL WINAPI SetThreadContext(  
    HANDLE hThread,  
    CONST CONTEXT *lpContext  
);
```

由于 `GetThreadContext` 和 `SetThreadContext` 可以操作 CPU 内部的寄存器，因此在一些高级技巧的编程中有一定应用。譬如，调试器可利用 `GetThreadContext` 挂起被调试线程获取其上下文，并设置上下文中的标志寄存器中的陷阱标志位，最后通过 `SetThreadContext` 使设置生效来进行单步调试。

8.实例

以下程序使用 `CreateThread` 创建两个线程，在这两个线程中 `Sleep` 一段时间，主线程通过 `GetExitCodeThread` 来判断两个线程是否结束运行：

```
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <conio.h>

DWORD WINAPI ThreadFunc(LPVOID);

int main()
{
    HANDLE hThrd1;
    HANDLE hThrd2;
    DWORD exitCode1 = 0;
    DWORD exitCode2 = 0;
    DWORD threadId;

    hThrd1 = CreateThread(NULL, 0, ThreadFunc, (LPVOID)1, 0, &threadId);
    if (hThrd1)
        printf("Thread 1 launched\n");

    hThrd2 = CreateThread(NULL, 0, ThreadFunc, (LPVOID)2, 0, &threadId);
    if (hThrd2)
        printf("Thread 2 launched\n");

    // Keep waiting until both calls to GetExitCodeThread succeed AND
    // neither of them returns STILL_ACTIVE.
    for (;;)
    {
        printf("Press any key to exit.\n");
        getch();

        GetExitCodeThread(hThrd1, &exitCode1);
        GetExitCodeThread(hThrd2, &exitCode2);
        if ( exitCode1 == STILL_ACTIVE )
            puts("Thread 1 is still running!");
        if ( exitCode2 == STILL_ACTIVE )
```

```

        puts("Thread 2 is still running!");
        if ( exitCode1 != STILL_ACTIVE && exitCode2 != STILL_ACTIVE )
            break;
    }

    CloseHandle(hThrd1);
    CloseHandle(hThrd2);

    printf("Thread 1 returned %d\n", exitCode1);
    printf("Thread 2 returned %d\n", exitCode2);

    return EXIT_SUCCESS;
}

/*
 * Take the startup value, do some simple math on it,
 * and return the calculated value.
 */
DWORD WINAPI ThreadFunc(LPVOID n)
{
    Sleep((DWORD)n*1000*2);
    return (DWORD)n * 10;
}

```

通过下面的程序我们可以看出多线程程序运行顺序的难以预料以及 WINAPI 的 CreateThread 函数与 C 运行时库的 _beginthread 的差别:

```

#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

DWORD WINAPI ThreadFunc(LPVOID);

int main()
{
    HANDLE hThrd;
    DWORD threadId;
    int i;

    for (i = 0; i < 5; i++)
    {
        hThrd = CreateThread(NULL, 0, ThreadFunc, (LPVOID)i, 0, &threadId);
        if (hThrd)

```

```

    {
        printf("Thread launched %d\n", i);
        CloseHandle(hThrd);
    }
}

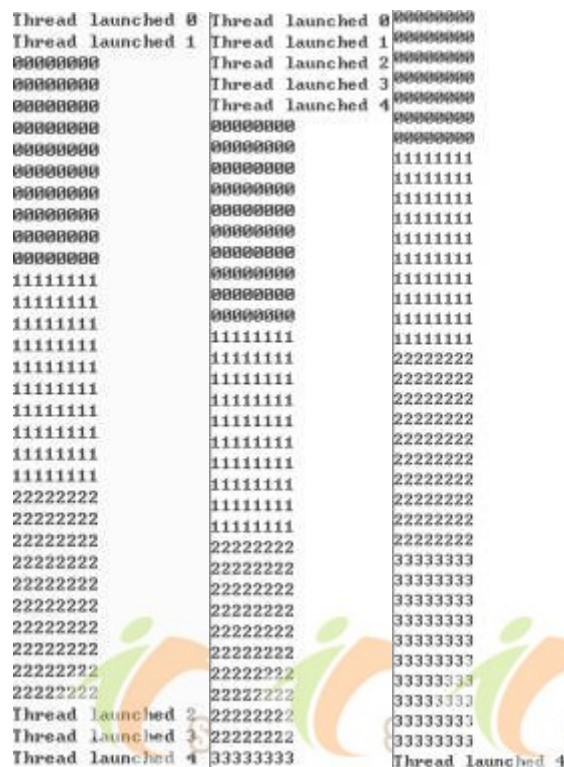
// Wait for the threads to complete.
Sleep(2000);

return EXIT_SUCCESS;
}

DWORD WINAPI ThreadFunc(LPVOID n)
{
    int i;
    for (i = 0; i < 10; i++)
        printf("%d%d%d%d%d%d%d%d\n", n, n, n, n, n, n, n, n);
    return 0;
}

```

运行的输出具有很大的随机性，这里摘取了几次结果的一部分（几乎每一次都不同）：



```

Thread launched 0 Thread launched 0 00000000
Thread launched 1 Thread launched 1 00000000
00000000 Thread launched 2 00000000
00000000 Thread launched 3 00000000
00000000 Thread launched 4 00000000
00000000 00000000 00000000
00000000 00000000 11111111
00000000 00000000 11111111
00000000 00000000 11111111
00000000 00000000 11111111
00000000 00000000 11111111
00000000 00000000 11111111
00000000 00000000 11111111
11111111 00000000 11111111
11111111 00000000 11111111
11111111 00000000 11111111
11111111 11111111 11111111
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
11111111 11111111 22222222
22222222 11111111 22222222
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
22222222 22222222 33333333
Thread launched 2 22222222 33333333
Thread launched 3 22222222 33333333
Thread launched 4 33333333 Thread launched 4

```

如果我们使用标准 C 库函数而不是多线程版的运行时库，则程序可能输出"3333444444"这样的结果，而使用多线程运行时库后，则可避免这一问题。

下列程序在主线程中创建一个 SecondThread，在 SecondThread 线程中通过自增对 Counter 计数到 1000000，主线程一直等待其结束：

```
#include <Win32.h>
#include <stdio.h>
#include <process.h>

unsigned Counter;
unsigned __stdcall SecondThreadFunc(void *pArguments)
{
    printf("In second thread...\n");

    while (Counter < 1000000)
        Counter++;

    _endthreadex(0);
    return 0;
}

int main()
{
    HANDLE hThread;
    unsigned threadID;

    printf("Creating second thread...\n");

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex(NULL, 0, &SecondThreadFunc, NULL, 0, &threadID);

    // Wait until second thread terminates
    WaitForSingleObject(hThread, INFINITE);
    printf("Counter should be 1000000; it is-> %d\n", Counter);

    // Destroy the thread object.
    CloseHandle(hThread);
}
```

共 3 页。 

深入浅出 Win32 多线程程序设计之线程通信

简介

线程之间通信的两个基本问题是互斥和同步。

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程互斥是指对于共享的操作系统资源（指的是广义的“资源”，而不是 Windows 的 .res 文件，譬如全

局变量就是一种共享资源），在各线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

线程互斥是一种特殊的线程同步。

实际上，互斥和同步对应着线程间通信发生的两种情况：

- (1) 当有多个线程访问共享资源而不使资源被破坏时；
- (2) 当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

在 WIN32 中，同步机制主要有以下几种：

- (1) 事件(Event);
- (2) 信号量(semaphore);
- (3) 互斥量(mutex);
- (4) 临界区(Critical section)。

全局变量

因为进程中的所有线程均可以访问所有的全局变量，因而全局变量成为 Win32 多线程通信的最简单方式。例如：

```
int var; //全局变量

UINT ThreadFunction(LPVOID pParam)
{
    var = 0;
    while (var < MaxValue)
    {
        //线程处理
        ::InterlockedIncrement(long*) &var;
    }
    return 0;
}

请看下列程序：
int globalFlag = false;
DWORD WINAPI ThreadFunc(LPVOID n)
{
    Sleep(2000);
    globalFlag = true;

    return 0;
}

int main()
{
    HANDLE hThrd;
    DWORD threadId;

    hThrd = CreateThread(NULL, 0, ThreadFunc, NULL, 0, &threadId);
```

```

        if (hThrd)
        {
            printf("Thread launched\n");
            CloseHandle(hThrd);
        }

        while (!globalFlag)
        ;
        printf("exit\n");
    }

```

上述程序中使用全局变量和 `while` 循环查询进行线程间同步，实际上，这是一种应该避免的方法，因为：

(1) 当主线程必须使自己与 `ThreadFunc` 函数的完成运行实现同步时，它并没有使自己进入睡眠状态。由于主线程没有进入睡眠状态，因此操作系统继续为它调度 C P U 时间，这就要占用其他线程的宝贵时间周期；

(2) 当主线程的优先级高于执行 `ThreadFunc` 函数的线程时，就会发生 `globalFlag` 永远不能被赋值为 `true` 的情况。因为在这种情况下，系统决不会将任何时间片分配给 `ThreadFunc` 线程。

事件

事件(Event)是 WIN32 提供的最灵活的线程间同步方式，事件可以处于激发状态(signaled or true)或未激发状态(unsignal or false)。根据状态变迁方式的不同，事件可分为两类：

(1) 手动设置：这种对象只可能用程序手动设置，在需要该事件或者事件发生时，采用 `SetEvent` 及 `ResetEvent` 来进行设置。

(2) 自动恢复：一旦事件发生并被处理后，自动恢复到没有事件状态，不需要再次设置。

创建事件的函数原型为：

```

HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    // SECURITY_ATTRIBUTES 结构指针，可为 NULL
    BOOL bManualReset,
    // 手动/自动
    // TRUE: 在 WaitForSingleObject 后必须手动调用 ResetEvent 清除信号
    // FALSE: 在 WaitForSingleObject 后，系统自动清除事件信号
    BOOL bInitialState, //初始状态
    LPCTSTR lpName //事件的名称
);

```

使用"事件"机制应注意以下事项：

(1) 如果跨进程访问事件，必须对事件命名，在对事件命名的时候，要注意不要与系统命名空间中的其它全局命名对象冲突；

(2) 事件是否要自动恢复；

(3) 事件的初始状态设置。

由于 `event` 对象属于内核对象，故进程 B 可以调用 `OpenEvent` 函数通过对象的名字获得进程 A 中 `event`

对象的句柄，然后将这个句柄用于 `ResetEvent`、`SetEvent` 和 `WaitForMultipleObjects` 等函数中。此法可以实现一个进程的线程控制另一进程中线程的运行，例如：

```
HANDLE hEvent=OpenEvent(EVENT_ALL_ACCESS,true,"MyEvent");  
ResetEvent(hEvent);
```

临界区

定义临界区变量

```
CRITICAL_SECTION gCriticalSection;
```

通常情况下，`CRITICAL_SECTION` 结构体应该被定义为全局变量，以便于进程中的所有线程方便地按照变量名来引用该结构体。

初始化临界区

```
VOID WINAPI InitializeCriticalSection(  
LPCRITICAL_SECTION lpCriticalSection  
//指向程序员定义的 CRITICAL_SECTION 变量  
);
```

该函数用于对 `pcs` 所指的 `CRITICAL_SECTION` 结构体进行初始化。该函数只是设置了一些成员变量，它的运行一般不会失败，因此它采用了 `VOID` 类型的返回值。该函数必须在任何线程调用 `EnterCriticalSection` 函数之前被调用，如果一个线程试图进入一个未初始化的 `CRITICAL_SECTION`，那么结果将是很难预计的。

删除临界区

```
VOID WINAPI DeleteCriticalSection(  
LPCRITICAL_SECTION lpCriticalSection  
//指向一个不再需要的 CRITICAL_SECTION 变量  
);
```

进入临界区

```
VOID WINAPI EnterCriticalSection(  
LPCRITICAL_SECTION lpCriticalSection  
//指向一个你即将锁定的 CRITICAL_SECTION 变量  
);
```

离开临界区

```
VOID WINAPI LeaveCriticalSection(  
LPCRITICAL_SECTION lpCriticalSection
```

```
//指向一个你即将离开的 CRITICAL_SECTION 变量
);
```

使用临界区编程的一般方法是：

```
void UpdateData()
{
    EnterCriticalSection(&gCriticalSection);
    ...//do something
    LeaveCriticalSection(&gCriticalSection);
}
```

关于临界区的使用，有下列注意点：

- (1) 每个共享资源使用一个 **CRITICAL_SECTION** 变量；
- (2) 不要长时间运行关键代码段，当一个关键代码段长时间运行时，其他线程就会进入等待状态，这会降低应用程序的运行性能；
- (3) 如果需要同时访问多个资源，则可能连续调用 **EnterCriticalSection**；
- (4) **Critical Section** 不是 OS 核心对象，如果进入临界区的线程"挂"了，将无法释放临界资源。这个缺点在 **Mutex** 中得到了弥补。

互斥

互斥量的作用是保证每次只能有一个线程获得互斥量而得以继续执行，使用 **CreateMutex** 函数创建：

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    // 安全属性结构指针，可为 NULL
    BOOL bInitialOwner,
    //是否占有该互斥量，TRUE：占有，FALSE：不占有
    LPCTSTR lpName
    //信号量的名称
);
```

Mutex 是核心对象，可以跨进程访问，下面的代码给出了从另一进程访问命名 **Mutex** 的例子：

```
HANDLE hMutex;
hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, L"mutexName");
if (hMutex){
    ...
}
else{
    ...
}
```

相关 API:

```
BOOL WINAPI ReleaseMutex(  
    HANDLE hMutex  
);
```

使用互斥编程的一般方法是:

```
void UpdateResource()  
{  
    WaitForSingleObject(hMutex,...);  
    ...//do something  
    ReleaseMutex(hMutex);  
}
```

互斥(mutex)内核对象能够确保线程拥有对单个资源的互斥访问权。互斥对象的行为特性与临界区相同,但是互斥对象属于内核对象,而临界区则属于用户方式对象,因此这导致 mutex 与 Critical Section 的如下不同:

- (1) 互斥对象的运行速度比关键代码段要慢;
- (2) 不同进程中的多个线程能够访问单个互斥对象;
- (3) 线程在等待访问资源时可以设定一个超时值。

下图更详细地列出了互斥与临界区的不同:

特性	互斥	临界区
运行速度	慢	快
是否能够跨进程	是	否
边界来使用		
声明	HANDLE hmtx;	CRITICAL_SECTION cs;
初始化	hmtx=CreateMutex (NULL, FALSE, NULL);	InitializeCriticalSection(&cs);
清除	CloseHandle(hmtx);	DeleteCriticalSection(&cs);
无限等待	WaitForSingleObject (hmtx,INFINITE);	EnterCriticalSection(&cs);
0等待	WaitForSingleObject (hmtx,0);	TryEnterCriticalSection(&cs);
任意等待	WaitForSingleObject (hmtx,dwMilliseconds);	不能
释放	ReleaseMutex(hmtx);	LeaveCriticalSection(&cs);
是否能够等待	是	否
其他内核对象	使用WaitForMultipleObjects或类似的函数)	

信号量

信号量是维护 0 到指定最大值之间的同步对象。信号量状态在其计数大于 0 时是有信号的,而其计数是 0 时是无信号的。信号量对象在控制上可以支持有限数量共享资源的访问。

信号量的特点和用途可用下列几句话定义：

- (1) 如果当前资源的数量大于 0，则信号量有效；
- (2) 如果当前资源数量是 0，则信号量无效；
- (3) 系统决不允许当前资源的数量为负值；
- (4) 当前资源数量决不能大于最大资源数量。

创建信号量

```
HANDLE CreateSemaphore (  
    PSECURITY_ATTRIBUTE psa,  
    LONG lInitialCount, //开始时可供使用的资源数  
    LONG lMaximumCount, //最大资源数  
    PCTSTR pszName);
```

释放信号量

通过调用 **ReleaseSemaphore** 函数，线程就能够对信标的当前资源数量进行递增，该函数原型为：

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lReleaseCount, //信号量的当前资源数增加 lReleaseCount  
    LPLONG lpPreviousCount  
);
```

打开信号量

和其他核心对象一样，信号量也可以通过名字跨进程访问，打开信号量的 API 为：

```
HANDLE OpenSemaphore (  
    DWORD fdwAccess,  
    BOOL bInherithandle,  
    PCTSTR pszName  
);
```

互锁访问

当必须以原子操作方式来修改单个值时，互锁访问函数是相当有用的。所谓原子访问，是指线程在访问资源时能够确保所有其他线程都不在同一时间内访问相同的资源。

请看下列代码：

```
int globalVar = 0;  
  
DWORD WINAPI ThreadFunc1(LPVOID n)  
{  
    globalVar++;  
}
```

```

        return 0;
    }

    DWORD WINAPI ThreadFunc2(LPVOID n)
    {
        globalVar++;
        return 0;
    }

```

运行 ThreadFunc1 和 ThreadFunc2 线程，结果是不可预料的，因为 globalVar++ 并不对应着一条机器指令，我们看看 globalVar++ 的反汇编代码：

```

00401038 mov eax,[globalVar (0042d3f0)]
0040103D add eax,1
00401040 mov [globalVar (0042d3f0)],eax

```

在 "mov eax,[globalVar (0042d3f0)]" 指令与 "add eax,1" 指令以及 "add eax,1" 指令与 "mov [globalVar (0042d3f0)],eax" 指令之间都可能发生线程切换，使得程序的执行后 globalVar 的结果不能确定。我们可以使用 InterlockedExchangeAdd 函数解决这个问题：

```

int globalVar = 0;

DWORD WINAPI ThreadFunc1(LPVOID n)
{
    InterlockedExchangeAdd(&globalVar,1);
    return 0;
}

DWORD WINAPI ThreadFunc2(LPVOID n)
{
    InterlockedExchangeAdd(&globalVar,1);
    return 0;
}

```

InterlockedExchangeAdd 保证对变量 globalVar 的访问具有"原子性"。互锁访问的控制速度非常快，调用一个互锁函数的 CPU 周期通常小于 50，不需要进行用户方式与内核方式的切换（该切换通常需要运行 1000 个 CPU 周期）。

互锁访问函数的缺点在于其只能对单一变量进行原子访问，如果要访问的资源比较复杂，仍要使用临界区或互斥。

可等待定时器

可等待定时器是在某个时间或按规定的间隔时间发出自己的信号通知的内核对象。它们通常用来在某个时间执行某个操作。

创建可等待定时器

```
HANDLE CreateWaitableTimer(  
    PSECURITY_ATTRIBUTES psa,  
    BOOL fManualReset, //人工重置或自动重置定时器  
    PCTSTR pszName);
```

设置可等待定时器

可等待定时器对象在非激活状态下被创建，程序员应调用 `SetWaitableTimer` 函数来界定定时器在何时被激活：

```
BOOL SetWaitableTimer(  
    HANDLE hTimer, //要设置的定时器  
    const LARGE_INTEGER *pDueTime, //指明定时器第一次激活的时间  
    LONG lPeriod, //指明此后定时器应该间隔多长时间激活一次  
    PTIMERAPCROUTINE pfnCompletionRoutine,  
    PVOID PvArgToCompletionRoutine,  
    BOOL fResume);
```

取消可等待定时器

```
BOOL Cancel WaitableTimer(  
    HANDLE hTimer //要取消的定时器  
);
```

打开可等待定时器

作为一种内核对象，`WaitableTimer` 也可以被其他进程以名字打开：

```
HANDLE OpenWaitableTimer (  
    DWORD fdwAccess,  
    BOOL bInheritable,  
    PCTSTR pszName  
);
```

实例

下面给出的一个程序可能发生死锁现象：

```
#include <windows.h>  
#include <stdio.h>  
CRITICAL_SECTION cs1, cs2;  
long WINAPI ThreadFn(long);  
main()  
{  
    long iThreadID;  
    InitializeCriticalSection(&cs1);
```



```

        InitializeCriticalSection(&cs2);
    CloseHandle(CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadFn, NULL,
        0,&iThreadID));
        while (TRUE)
        {
            EnterCriticalSection(&cs1);
            printf("\n 线程 1 占用临界区 1");
            EnterCriticalSection(&cs2);
            printf("\n 线程 1 占用临界区 2");

            printf("\n 线程 1 占用两个临界区");

            LeaveCriticalSection(&cs2);
            LeaveCriticalSection(&cs1);

            printf("\n 线程 1 释放两个临界区");
            Sleep(20);
        };
        return (0);
    }

long WINAPI ThreadFn(long lParam)
    {
        while (TRUE)
        {
            EnterCriticalSection(&cs2);
            printf("\n 线程 2 占用临界区 2");
            EnterCriticalSection(&cs1);
            printf("\n 线程 2 占用临界区 1");

            printf("\n 线程 2 占用两个临界区");

            LeaveCriticalSection(&cs1);
            LeaveCriticalSection(&cs2);

            printf("\n 线程 2 释放两个临界区");
            Sleep(20);
        };
    }

```

运行这个程序，在中途一旦发生这样的输出：

线程 1 占用临界区 1

线程 2 占用临界区 2

或

线程 2 占用临界区 2

线程 1 占用临界区 1

或

线程 1 占用临界区 2

线程 2 占用临界区 1

或

线程 2 占用临界区 1

线程 1 占用临界区 2

程序就“死”掉了，再也运行不下去。因为这样的输出，意味着两个线程相互等待对方释放临界区，也即出现了死锁。

如果我们将线程 2 的控制函数改为：

```
long WINAPI ThreadFn(long lParam)
{
    while (TRUE)
    {
        EnterCriticalSection(&cs1);
        printf("\n 线程 2 占用临界区 1");
        EnterCriticalSection(&cs2);
        printf("\n 线程 2 占用临界区 2");

        printf("\n 线程 2 占用两个临界区");

        LeaveCriticalSection(&cs1);
        LeaveCriticalSection(&cs2);

        printf("\n 线程 2 释放两个临界区");
        Sleep(20);
    }
}
```

再次运行程序，死锁被消除，程序不再挡掉。这是因为我们改变了线程 2 中获得临界区 1、2 的顺序，消除了线程 1、2 相互等待资源的可能性。

由此我们得出结论，在使用线程间的同步机制时，要特别留心死锁的发生。

深入浅出 Win32 多线程设计之 MFC 的多线程

1、创建和终止线程

在 MFC 程序中创建一个线程，宜调用 `AfxBeginThread` 函数。该函数因参数不同而具有两种重载版本，分别对应工作者线程和用户接口（UI）线程。

工作者线程

```
CWinThread *AfxBeginThread(
    AFX_THREADPROC pfnThreadProc, //控制函数
```

```

        LPVOID pParam, //传递给控制函数的参数
        int nPriority = THREAD_PRIORITY_NORMAL, //线程的优先级
        UINT nStackSize = 0, //线程的堆栈大小
        DWORD dwCreateFlags = 0, //线程的创建标志
        LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL //线程的安全属性
    );

```

工作者线程编程较为简单，只需编写线程控制函数和启动线程即可。下面的代码给出了定义一个控制函数和启动它的过程：

```

//线程控制函数
UINT MfcThreadProc(LPVOID lpParam)
{
    CExampleClass *lpObject = (CExampleClass*)lpParam;
    if (lpObject == NULL || !lpObject->IsKindof(RUNTIME_CLASS(CExampleClass)))
        return -1; //输入参数非法
    //线程成功启动
    while (1)
    {
        ...//
    }
    return 0;
}

//在 MFC 程序中启动线程
AfxBeginThread(MfcThreadProc, lpObject);

```

UI 线程

创建用户界面线程时，必须首先从 `CWinThread` 派生类，并使用 `DECLARE_DYNCREATE` 和 `IMPLEMENT_DYNCREATE` 宏声明此类。

下面给出了 `CWinThread` 类的原型（添加了关于其重要函数功能和是否需要被继承类重载的注释）：

```

class CWinThread : public CCmdTarget
{
    DECLARE_DYNAMIC(CWinThread)

public:
    // Constructors
    CWinThread();

    BOOL CreateThread(DWORD dwCreateFlags = 0, UINT nStackSize = 0,
        LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

    // Attributes

```

```

CWnd* m_pMainWnd; // main window (usually same AfxGetApp()->m_pMainWnd)
CWnd* m_pActiveWnd; // active main window (may not be m_pMainWnd)
BOOL m_bAutoDelete; // enables 'delete this' after thread termination

// only valid while running
HANDLE m_hThread; // this thread's HANDLE
operator HANDLE() const;
DWORD m_nThreadID; // this thread's ID

int GetThreadPriority();
BOOL SetThreadPriority(int nPriority);

// Operations
DWORD SuspendThread();
DWORD ResumeThread();
BOOL PostThreadMessage(UINT message, WPARAM wParam, LPARAM lParam);

// Overridables
//执行线程实例初始化，必须重写
virtual BOOL InitInstance();

// running and idle processing
//控制线程的函数，包含消息泵，一般不重写
virtual int Run();

//消息调度到 TranslateMessage 和 DispatchMessage 之前对其进行筛选，
//通常不重写
virtual BOOL PreTranslateMessage(MSG* pMsg);

virtual BOOL PumpMessage(); // low level message pump

//执行线程特定的闲置时间处理，通常不重写
virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
virtual BOOL IsIdleMessage(MSG* pMsg); // checks for special messages

//线程终止时执行清除，通常需要重写
virtual int ExitInstance(); // default will 'delete this'

//截获由线程的消息和命令处理程序引发的未处理异常，通常不重写
virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);

// Advanced: handling messages sent to message filter hook
virtual BOOL ProcessMessageFilter(int code, LPMSG lpMsg);

```

```

// Advanced: virtual access to m_pMainWnd
virtual CWnd* GetMainWnd();

// Implementation
public:
virtual ~CWinThread();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
int m_nDisablePumpCount; // Diagnostic trap to detect illegal re-entrancy
#endif
void CommonConstruct();
virtual void Delete();
// 'delete this' only if m_bAutoDelete == TRUE

// message pump for Run
MSG m_msgCur; // current message

public:
// constructor used by implementation of AfxBeginThread
CWinThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam);

// valid after construction
LPVOID m_pThreadParams; // generic parameters passed to starting function
AFX_THREADPROC m_pfnThreadProc;

// set after OLE is initialized
void (AFXAPI* m_lpfnOleTermOrFreeLib)(BOOL, BOOL);
COleMessageFilter* m_pMessageFilter;

protected:
CPoint m_ptCursorLast; // last mouse position
UINT m_nMsgLast; // last mouse message
BOOL DispatchThreadMessageEx(MSG* msg); // helper
void DispatchThreadMessage(MSG* msg); // obsolete
};

```

启动 UI 线程的 AfxBeginThread 函数的原型为：

```

CWinThread *AfxBeginThread(
//从 CWinThread 派生的类的  RUNTIME_CLASS
CRuntimeClass *pThreadClass,
int nPriority = THREAD_PRIORITY_NORMAL,
UINT nStackSize = 0,

```

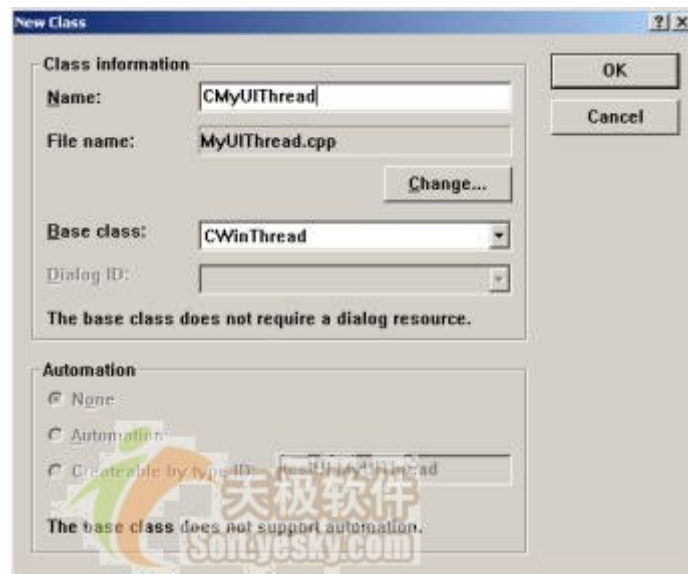
```

        DWORD dwCreateFlags = 0,
        LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
    );

```

我们可以方便地使用 VC++ 6.0 类向导定义一个继承自 CWinThread 的用户线程类。下面给出产生我们自定义的 CWinThread 子类 CMyUIThread 的方法。

打开 VC++ 6.0 类向导，在如下窗口中选择 Base Class 类为 CWinThread，输入子类名为 CMyUIThread，点击"OK"按钮后就产生了类 CMyUIThread。



其源代码框架为：

```

////////////////////////////////////
// CMyUIThread thread

class CMyUIThread : public CWinThread
{
    DECLARE_DYNCREATE(CMyUIThread)
protected:
    CMyUIThread(); // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides

```

```

//{{AFX_VIRTUAL(CMyUIThread)

public:
virtual BOOL InitInstance();
virtual int ExitInstance();
//}}AFX_VIRTUAL

// Implementation
protected:
virtual ~CMyUIThread();

// Generated message map functions
//{{AFX_MSG(CMyUIThread)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG

DECLARE_MESSAGE_MAP()
};

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

// CMyUIThread

IMPLEMENT_DYNCREATE(CMyUIThread, CWinThread)

CMyUIThread::CMyUIThread()
{

}

CMyUIThread::~CMyUIThread()
{

}

BOOL CMyUIThread::InitInstance()
{
// TODO: perform and per-thread initialization here
return TRUE;
}

int CMyUIThread::ExitInstance()
{
// TODO: perform any per-thread cleanup here
return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP(CMyUIThread, CWinThread)
//{{AFX_MSG_MAP(CMyUIThread)
// NOTE - the ClassWizard will add and remove mapping macros here.

```

```
//}}AFX_MSG_MAP  
END_MESSAGE_MAP()
```

使用下列代码就可以启动这个 UI 线程：

```
CMyUIThread *pThread;  
pThread = (CMyUIThread*)  
AfxBeginThread( RUNTIME_CLASS(CMyUIThread) );
```

另外，我们也可以不用 `AfxBeginThread` 创建线程，而是分如下两步完成：

- (1) 调用线程类的构造函数创建一个线程对象；
- (2) 调用 `CWinThread::CreateThread` 函数来启动该线程。

在线程自身内调用 `AfxEndThread` 函数可以终止该线程：

```
void AfxEndThread(  
    UINT nExitCode //the exit code of the thread  
);
```

对于 UI 线程而言，如果消息队列中放入了 `WM_QUIT` 消息，将结束线程。

关于 UI 线程和工作者线程的分配，最好的做法是：将所有与 UI 相关的操作放入主线程，其它的纯粹的运算工作交给独立的数个工作者线程。

候捷先生早些时间喜欢为 MDI 程序的每个窗口创建一个线程，他后来澄清了这个错误。因为如果为 MDI 程序的每个窗口都单独创建一个线程，在窗口进行切换的时候，将进行线程的上下文切换！

2. 线程间通信

MFC 中定义了继承自 `CSyncObject` 类的 `CCriticalSection`、`CEvent`、`CMutex`、`CSemaphore` 类封装和简化了 WIN32 API 所提供的临界区、事件、互斥和信号量。使用这些同步机制，必须包含 "afxmt.h" 头文件。下图给出了类的继承关系：



作为 `CSyncObject` 类的继承类，我们仅仅使用基类 `CSyncObject` 的接口函数就可以方便、统一的操作 `CCriticalSection`、`CEvent`、`CMutex`、`CSemaphore` 类，下面是 `CSyncObject` 类的原型：

```
class CSyncObject : public CObject  
{
```



```

DECLARE_DYNAMIC(CSyncObject)

// Constructor
public:
CSyncObject(LPCTSTR pstrName);

// Attributes
public:
operator HANDLE() const;
HANDLE m_hObject;

// Operations
virtual BOOL Lock(DWORD dwTimeout = INFINITE);
virtual BOOL Unlock() = 0;
virtual BOOL Unlock(LONG /* lCount */, LPLONG /* lpPrevCount=NULL */)
{ return TRUE; }

// Implementation
public:
virtual ~CSyncObject();
#ifdef _DEBUG
CString m_strName;
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
friend class CSingleLock;
friend class CMultiLock;
};

```

CSyncObject 类最主要的两个函数是 Lock 和 Unlock，若我们直接使用 CSyncObject 类及其派生类，我们需要非常小心地在 Lock 之后调用 Unlock。

MFC 提供的另两个类 CSingleLock（等待一个对象）和 CMultiLock（等待多个对象）为我们编写应用程序提供了更灵活的机制，下面以实际来阐述 CSingleLock 的用法：

```

class CThreadSafeWnd
{
public:
    CThreadSafeWnd(){}
    ~CThreadSafeWnd(){}
    void SetWindow(CWnd *pwnd)
    {
        m_pCWnd = pwnd;
    }
}

```

```

void PaintBall(COLORREF color, CRect &rc);

private:
    CWnd *m_pCWnd;
    CCriticalSection m_CSect;
};

void CThreadSafeWnd::PaintBall(COLORREF color, CRect &rc)
{
    CSingleLock csl(&m_CSect);
    //缺省的 Timeout 是 INFINITE，只有 m_CSect 被激活，csl.Lock()才能返回
    //true，这里一直等待
    if (csl.Lock())
    {
        // not necessary
        //AFX_MANAGE_STATE(AfxGetStaticModuleState( ));
        CDC *pdc = m_pCWnd->GetDC();
        CBrush brush(color);
        CBrush *oldbrush = pdc->SelectObject(&brush);
        pdc->Ellipse(rc);
        pdc->SelectObject(oldbrush);
        GdiFlush(); // don't wait to update the display
    }
}

```

上述实例讲述了用 CSingleLock 对 Windows GDI 相关对象进行保护的方法，下面再给出一个其他方面的例子：

```

int array1[10], array2[10];
CMutexSection section; //创建一个 CMutex 类的对象

//赋值线程控制函数
UINT EvaluateThread(LPVOID param)
{
    CSingleLock singlelock;
    singlelock(&section);

    //互斥区域
    singlelock.Lock();
    for (int i = 0; i < 10; i++)
        array1[i] = i;
    singlelock.Unlock();
}

//拷贝线程控制函数

```

```

UINT CopyThread(LPVOID param)
{
    CSingleLock singlelock;
    singlelock(&section);

    //互斥区域
    singlelock.Lock();
    for (int i = 0; i < 10; i++)
        array2[i] = array1[i];
    singlelock.Unlock();
}

AfxBeginThread(EvaluateThread, NULL); //启动赋值线程
AfxBeginThread(CopyThread, NULL); //启动拷贝线程

```

上面的例子中启动了两个线程 EvaluateThread 和 CopyThread，线程 EvaluateThread 把 10 个数赋值给数组 array1[]，线程 CopyThread 将数组 array1[] 拷贝给数组 array2[]。由于数组的拷贝和赋值都是整体行为，如果不以互斥形式执行代码段：

```

for (int i = 0; i < 10; i++)
    array1[i] = i;

```

和

```

for (int i = 0; i < 10; i++)
    array2[i] = array1[i];

```

其结果是很难预料的！

除了可使用 CCriticalSection、CEvent、CMutex、CSemaphore 作为线程间同步通信的方式以外，我们还可以利用 PostThreadMessage 函数在线程间发送消息：

```

BOOL PostThreadMessage(DWORD idThread, // thread identifier
    UINT Msg, // message to post
    WPARAM wParam, // first message parameter
    LPARAM lParam // second message parameter
);

```

3. 线程与消息队列

在 WIN32 中，每一个线程都对应着一个消息队列。由于一个线程可以产生数个窗口，所以并不是每个窗口都对应着一个消息队列。下列几句话应该作为"定理"被记住：

"定理" 一

所有产生给某个窗口的消息，都先由创建这个窗口的线程处理；

"定理" 二

Windows 屏幕上的每一个控件都是一个窗口，有对应的窗口函数。

消息的发送通常有两种方式，一是 `SendMessage`，一是 `PostMessage`，其原型分别为：

```
LRESULT SendMessage(HWND hWnd, // handle of destination window
                    UINT Msg, // message to send
                    WPARAM wParam, // first message parameter
                    LPARAM lParam // second message parameter
                    );
BOOL PostMessage(HWND hWnd, // handle of destination window
                UINT Msg, // message to post
                WPARAM wParam, // first message parameter
                LPARAM lParam // second message parameter
                );
```

两个函数原型中的四个参数的意义相同，但是 `SendMessage` 和 `PostMessage` 的行为有差异。`SendMessage` 必须等待消息被处理后才返回，而 `PostMessage` 仅仅将消息放入消息队列。`SendMessage` 的目标窗口如果属于另一个线程，则会发生线程上下文切换，等待另一线程处理完成消息。为了防止另一线程当掉，导致 `SendMessage` 永远不能返回，我们可以调用 `SendMessageTimeout` 函数：

```
LRESULT SendMessageTimeout(
    HWND hWnd, // handle of destination window
    UINT Msg, // message to send
    WPARAM wParam, // first message parameter
    LPARAM lParam, // second message parameter
    UINT fuFlags, // how to send the message
    UINT uTimeout, // time-out duration
    LPDWORD lpdwResult // return value for synchronous call
    );
```

4. MFC 线程、消息队列与 MFC 程序的"生死因果"

分析 MFC 程序的主线程启动及消息队列处理的过程将有助于我们进一步理解 UI 线程与消息队列的关系，为此我们需要简单地叙述一下 MFC 程序的"生死因果"（侯捷：《深入浅出 MFC》）。

使用 VC++ 6.0 的向导完成一个最简单的单文档架构 MFC 应用程序 `MFCThread`：

- (1) 输入 MFC EXE 工程名 `MFCThread`；
- (2) 选择单文档架构，不支持 Document/View 结构；
- (3) `ActiveX`、`3D container` 等其他选项都选择无。

我们来分析这个工程。下面是产生的核心源代码：

`MFCThread.h` 文件

```

class CMFCThreadApp : public CWinApp
{
public:
    CMFCThreadApp();

    // Overrides
    // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CMFCThreadApp)
public:
    virtual BOOL InitInstance();
   //}}AFX_VIRTUAL

    // Implementation

public:
    //{{AFX_MSG(CMFCThreadApp)
    afx_msg void OnAppAbout();
    // NOTE - the ClassWizard will add and remove member functions here.
    // DO NOT EDIT what you see in these blocks of generated code !
   //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

MFCThread.cpp 文件

```

CMFCThreadApp theApp;

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

// CMFCThreadApp initialization

BOOL CMFCThreadApp::InitInstance()
{
    ...

    CMainFrame* pFrame = new CMainFrame;
    m_pMainWnd = pFrame;

    // create and load the frame with its resources
    pFrame->LoadFrame(IDR_MAINFRAME,WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
        NULL,NULL);

    // The one and only window has been initialized, so show and update it.
    pFrame->ShowWindow(SW_SHOW);
    pFrame->UpdateWindow();

    return TRUE;
}

```

```
}
```

MainFrm.h 文件

```
#include "ChildView.h"

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
protected:
    DECLARE_DYNAMIC(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CMainFrame)
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra, AFX_CMDHANDLERINFO*
    pHandlerInfo);
//}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
    CChildView m_wndView;

// Generated message map functions
protected:
    {{{AFX_MSG(CMainFrame)
    afx_msg void OnSetFocus(CWnd *pOldWnd);
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code!
}}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

MainFrm.cpp 文件

```
IMPLEMENT_DYNAMIC(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code !
    ON_WM_SETFOCUS()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

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

// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
    cs.lpszClass = AfxRegisterWndClass(0);
    return TRUE;
}
```

ChildView.h 文件

```
// CChildView window

class CChildView : public CWnd
{
    // Construction
public:
```

```

        CChildView();

        // Attributes
        public:
        // Operations
        public:
        // Overrides

        // ClassWizard generated virtual function overrides
        //{ {AFX_VIRTUAL(CChildView)
        protected:
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //} }AFX_VIRTUAL

        // Implementation
        public:
        virtual ~CChildView();

        // Generated message map functions
        protected:
        //{ {AFX_MSG(CChildView)
        afx_msg void OnPaint();
        //} }AFX_MSG
        DECLARE_MESSAGE_MAP()
    };

```

ChildView.cpp 文件

```

// CChildView

CChildView::CChildView()
{
}

CChildView::~~CChildView()
{
}

BEGIN_MESSAGE_MAP(CChildView,CWnd )
    //{ {AFX_MSG_MAP(CChildView)
    ON_WM_PAINT()
    //} }AFX_MSG_MAP
    END_MESSAGE_MAP()

////////////////////////////////////
// CChildView message handlers

```

```

BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)

```



```

        {
            if (!CWnd::PreCreateWindow(cs))
                return FALSE;

            cs.dwExStyle |= WS_EX_CLIENTEDGE;
            cs.style &= ~WS_BORDER;
            cs.lpszClass =
AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS,::LoadCursor(NULL,
                    IDC_ARROW),
                    HBRUSH(COLOR_WINDOW+1),NULL);

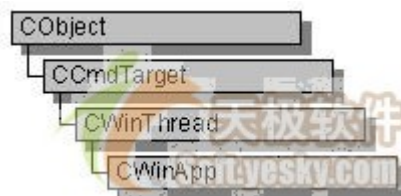
            return TRUE;
        }

void CChildView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
    // Do not call CWnd::OnPaint() for painting messages
}

```

文件 MFCThread.h 和 MFCThread.cpp 定义和实现的类 CMFCThreadApp 继承自 CWinApp 类，而 CWinApp 类又继承自 CWinThread 类（CWinThread 类又继承自 CCmdTarget 类），所以 CMFCThread 本质上是一个 MFC 线程类，下图给出了相关的类层次结构：



我们提取 CWinApp 类原型的一部分：

```

class CWinApp : public CWinThread
{
    DECLARE_DYNAMIC(CWinApp)

public:
    // Constructor
    CWinApp(LPCTSTR lpszAppName = NULL); // default app name

    // Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
}

```

```

        int m_nCmdShow;

        // Running args (can be changed in InitInstance)
        LPCTSTR m_pszAppName; // human readable name
        LPCTSTR m_pszExeName; // executable name (no spaces)
        LPCTSTR m_pszHelpFilePath; // default based on module path
        LPCTSTR m_pszProfileName; // default based on app name


        // Overridables
        virtual BOOL InitApplication();
        virtual BOOL InitInstance();
        virtual int ExitInstance(); // return app exit code
        virtual int Run();

        virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
        virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);


    public:
        virtual ~CWinApp();

    protected:
        DECLARE_MESSAGE_MAP()

    };

```

SDK 程序的 WinMain 所完成的工作现在由 CWinApp 的三个函数完成：

```

virtual BOOL InitApplication();
virtual BOOL InitInstance();
virtual int Run();

```

"CMFCThreadApp theApp;"语句定义的全局变量 theApp 是整个程式的 application object，每一个 MFC 应用程序都有一个。当我们执行 MFCThread 程序的时候，这个全局变量被构造。theApp 配置完成后，WinMain 开始执行。但是程序中并没有 WinMain 的代码，它在哪里呢？原来 MFC 早已准备好并由 Linker 直接加到应用程序代码中的，其原型为（存在于 VC++6.0 安装目录下提供的 APPMODUL.CPP 文件中）：

```

extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
          LPCTSTR lpCmdLine, int nCmdShow)
{
    // call shared/exported WinMain
    return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}

```

其中调用的 AfxWinMain 如下（存在于 VC++6.0 安装目录下提供的 WINMAIN.CPP 文件中）：

```

int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

```

```

LPTSTR lpCmdLine, int nCmdShow)
{
    ASSERT(hPrevInstance == NULL);

    int nReturnCode = -1;
    CWinThread* pThread = AfxGetThread();
    CWinApp* pApp = AfxGetApp();

    // AFX internal initialization
    if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
        goto InitFailure;

    // App global initializations (rare)
    if (pApp != NULL && !pApp->InitApplication())
        goto InitFailure;

    // Perform specific initializations
    if (!pThread->InitInstance())
    {
        if (pThread->m_pMainWnd != NULL)
        {
            TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
            pThread->m_pMainWnd->DestroyWindow();
        }
        nReturnCode = pThread->ExitInstance();
        goto InitFailure;
    }
    nReturnCode = pThread->Run();

    InitFailure:
    #ifdef _DEBUG
        // Check for missing AfxLockTempMap calls
        if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
        {
            TRACE1("Warning: Temp map lock count non-zero (%ld).\n",
                AfxGetModuleThreadState()->m_nTempMapLock);
        }
        AfxLockTempMaps();
        AfxUnlockTempMaps(-1);
    #endif

    AfxWinTerm();
    return nReturnCode;
}

```

我们提取主干，实际上，这个函数做的事情主要是：

```
CWinThread* pThread = AfxGetThread();
CWinApp* pApp = AfxGetApp();
AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
pApp->InitApplication()
pThread->InitInstance()
pThread->Run();
```

其中，`InitApplication` 是注册窗口类别的场所；`InitInstance` 是产生窗口并显示窗口的场所；`Run` 是提取并分派消息的场所。这样，MFC 就同 WIN32 SDK 程序对应起来了。`CWinThread::Run` 是程序生命的“活水源头”（侯捷：《深入浅出 MFC》，函数存在于 VC++ 6.0 安装目录下提供的 `THRD CORE.CPP` 文件中）：

```
// main running routine until thread exits
int CWinThread::Run()
{
    ASSERT_VALID(this);

    // for tracking the idle time state
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;

    // acquire and dispatch messages until a WM_QUIT message is received.
    for (;;)
    {
        // phase1: check to see if we can do idle work
        while (bIdle && !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }

        // phase2: pump messages while available
        do
        {
            // pump message, but quit on WM_QUIT
            if (!PumpMessage())
                return ExitInstance();

            // reset "no idle" state after pumping "normal" message
            if (IsIdleMessage(&m_msgCur))
            {
```

```

        bIdle = TRUE;
        lIdleCount = 0;
    }

} while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));

    }

    ASSERT(FALSE); // not reachable
}

```

其中的 PumpMessage 函数又对应于：

```

////////////////////////////////////
// CWinThread implementation helpers

BOOL CWinThread::PumpMessage()
{
    ASSERT_VALID(this);

    if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
    {
        return FALSE;
    }

    // process this message
    if(m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
    {
        ::TranslateMessage(&m_msgCur);
        ::DispatchMessage(&m_msgCur);
    }
    return TRUE;
}

```

因此，忽略 IDLE 状态，整个 RUN 的执行提取主干就是：

```

do {
    ::GetMessage(&msg,...);
    PreTranslateMessage(&msg);
    ::TranslateMessage(&msg);
    ::DispatchMessage(&msg);
    ...
} while (::PeekMessage(...));

```

由此，我们建立了 MFC 消息获取和派生机制与 WIN32 SDK 程序之间的对应关系。下面继续分析 MFC

消息的"绕行"过程。

在 MFC 中，只要是 CWnd 衍生类别，就可以拦下任何 Windows 消息。与窗口无关的 MFC 类别（例如 CDocument 和 CWinApp）如果也想处理消息，必须衍生自 CCmdTarget，并且只可能收到 WM_COMMAND 消息。所有能进行 MESSAGE_MAP 的类都继承自 CCmdTarget，如：



MFC 中 MESSAGE_MAP 的定义依赖于以下三个宏：

```
DECLARE_MESSAGE_MAP()

BEGIN_MESSAGE_MAP(
theClass, //Specifies the name of the class whose message map this is
baseClass //Specifies the name of the base class of theClass
)

END_MESSAGE_MAP()
```

我们程序中涉及到的有：MFCThread.h、MainFrm.h、ChildView.h 文件

```
DECLARE_MESSAGE_MAP()

MFCThread.cpp 文件
BEGIN_MESSAGE_MAP(CMFCThreadApp, CWinApp)
//{{AFX_MSG_MAP(CMFCThreadApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

MainFrm.cpp 文件
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code !
ON_WM_SETFOCUS()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

ChildView.cpp 文件
BEGIN_MESSAGE_MAP(CChildView, CWnd )
```

```

//{{AFX_MSG_MAP(CChildView)
    ON_WM_PAINT()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

由这些宏，MFC 建立了一个消息映射表（消息流动网），按照消息流动网匹配对应的消息处理函数，完成整个消息的"绕行"。

看到这里相信你有这样的疑问：程序定义了 CWinApp 类的 theApp 全局变量，可是从来没有调用 AfxBeginThread 或 theApp.CreateThread 启动线程呀，theApp 对应的线程是怎么启动的？

答：MFC 在这里用了很高明的一招。实际上，程序开始运行，第一个线程是由操作系统（OS）启动的，在 CWinApp 的构造函数里，MFC 将 theApp"对应"向了这个线程，具体的实现是这样的：

```

CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    if (lpszAppName != NULL)
        m_pszAppName = _tcsdup(lpszAppName);
    else
        m_pszAppName = NULL;

    // initialize CWinThread state
    AFX_MODULE_STATE *pModuleState = _AFX_CMDTARGT_GETSTATE();
    AFX_MODULE_THREAD_STATE *pThreadState = pModuleState->m_thread;
    ASSERT(AfxGetThread() == NULL);
    pThreadState->m_pCurrentWinThread = this;
    ASSERT(AfxGetThread() == this);
    m_hThread = ::GetCurrentThread();
    m_nThreadID = ::GetCurrentThreadId();

    // initialize CWinApp state
    ASSERT(afxCurrentWinApp == NULL); // only one CWinApp object please
    pModuleState->m_pCurrentWinApp = this;
    ASSERT(AfxGetApp() == this);

    // in non-running state until WinMain
    m_hInstance = NULL;
    m_pszHelpFilePath = NULL;
    m_pszProfileName = NULL;
    m_pszRegistryKey = NULL;
    m_pszExeName = NULL;
    m_pRecentFileList = NULL;
    m_pDocManager = NULL;

    m_atomApp = m_atomSystemTopic = NULL; //微软懒鬼？或者他认为

```

```

//这样连等含义更明确?
m_lpCmdLine = NULL;
m_pCmdInfo = NULL;

// initialize wait cursor state
m_nWaitCursorCount = 0;
m_hcurWaitCursorRestore = NULL;

// initialize current printer state
m_hDevMode = NULL;
m_hDevNames = NULL;
m_nNumPreviewPages = 0; // not specified (defaults to 1)

// initialize DAO state
m_lpfndaoTerm = NULL; // will be set if AfxDaoInit called

// other initialization
m_bHelpMode = FALSE;
m_nSafetyPoolSize = 512; // default size
}

```

很显然，theApp 成员变量都被赋予 OS 启动的这个当前线程相关的值，如代码：

```

m_hThread = ::GetCurrentThread();//theApp 的线程句柄等于当前线程句柄
m_nThreadId = ::GetCurrentThreadId();//theApp 的线程 ID 等于当前线程 ID

```

所以 CWinApp 类几乎只是为 MFC 程序的第一个线程量身定制的，它不需要也不能被 AfxBeginThread 或 theApp.CreateThread"再次"启动。这就是 CWinApp 类和 theApp 全局变量的内涵！如果你要再增加一个 UI 线程，不要继承类 CWinApp，而应继承类 CWinThread。而参考第 1 节，由于我们一般以主线程（在 MFC 程序里实际上就是 OS 启动的第一个线程）处理所有窗口的消息，所以我们几乎没有再启动 UI 线程的需求！

深入浅出 Win32 多线程程序设计之综合实例

本章我们将以工业控制和嵌入式系统中运用极为广泛的串口通信为例讲述多线程的典型应用。

而网络通信也是多线程应用最广泛的领域之一，所以本章的最后一节也将对多线程网络通信进行简短的描述。

1. 串口通信

在工业控制系统中，工控机（一般都基于 PC Windows 平台）经常需要与单片机通过串口进行通信。因此，操作和使用 PC 的串口成为大多数单片机、嵌入式系统领域工程师必须具备的能力。

串口的使用需要通过三个步骤来完成的：

- (1) 打开通信端口；

(2) 初始化串口，设置波特率、数据位、停止位、奇偶校验等参数。为了给读者一个直观的印象，下图从 Windows 的"控制面板->系统->设备管理器->通信端口 (COM1)"打开 COM 的设置窗口：



(3) 读写串口。

在 WIN32 平台下，对通信端口进行操作跟基本的文件操作一样。

创建/打开 COM 资源

下列函数如果调用成功，则返回一个标识通信端口的句柄，否则返回-1：

```
HANDLE CreateFile(PCTSTR lpFileName, //通信端口名，如"COM1"
                  WORD dwDesiredAccess, //对资源的访问类型
                  WORD dwShareMode, //指定共享模式，COM 不能共享，该参数为 0
                  PSECURITY_ATTRIBUTES lpSecurityAttributes,
                  //安全描述符指针，可为 NULL
                  WORD dwCreationDisposition, //创建方式
                  WORD dwFlagsAndAttributes, //文件属性，可为 NULL
                  HANDLE hTemplateFile //模板文件句柄，置为 NULL
                  );
```

获得/设置 COM 属性

下列函数可以获得 COM 口的设备控制块，从而获得相关参数：

```
BOOL WINAPI GetCommState(
    HANDLE hFile, //标识通信端口的句柄
    LPDCB lpDCB //指向一个设备控制块(DCB 结构)的指针
);
```

如果要调整通信端口的参数，则需要重新配置设备控制块，再用 WIN32 API SetCommState()函数进行设置：

```
BOOL SetCommState(  
    HANDLE hFile, //标识通信端口的句柄  
    LPDCB lpDCB //指向一个设备控制块(DCB 结构)的指针  
);
```

DCB 结构包含了串口的各项参数设置，如下：

```
typedef struct _DCB  
{  
    // dcb  
    DWORD DCBlength; // sizeof(DCB)  
    DWORD BaudRate; // current baud rate  
    DWORD fBinary: 1; // binary mode, no EOF check  
    DWORD fParity: 1; // enable parity checking  
    DWORD fOutxCtsFlow: 1; // CTS output flow control  
    DWORD fOutxDsrFlow: 1; // DSR output flow control  
    DWORD fDtrControl: 2; // DTR flow control type  
    DWORD fDsrSensitivity: 1; // DSR sensitivity  
    DWORD fTXContinueOnXoff: 1; // XOFF continues Tx  
    DWORD fOutX: 1; // XON/XOFF out flow control  
    DWORD fInX: 1; // XON/XOFF in flow control  
    DWORD fErrorChar: 1; // enable error replacement  
    DWORD fNull: 1; // enable null stripping  
    DWORD fRtsControl: 2; // RTS flow control  
    DWORD fAbortOnError: 1; // abort reads/writes on error  
    DWORD fDummy2: 17; // reserved  
    WORD wReserved; // not currently used  
    WORD XonLim; // transmit XON threshold  
    WORD XoffLim; // transmit XOFF threshold  
    BYTE ByteSize; // number of bits/byte, 4-8  
    BYTE Parity; // 0-4=no,odd,even,mark,space  
    BYTE StopBits; // 0,1,2 = 1, 1.5, 2  
    char XonChar; // Tx and Rx XON character  
    char XoffChar; // Tx and Rx XOFF character  
    char ErrorChar; // error replacement character  
    char EofChar; // end of input character  
    char EvtChar; // received event character  
    WORD wReserved1; // reserved; do not use  
} DCB;
```

读写串口

在读写串口之前，还要用 `PurgeComm()`函数清空缓冲区，并用 `SetCommMask()`函数设置事件掩模来监视指定通信端口上的事件，其原型为：

```
BOOL SetCommMask(  
    HANDLE hFile, //标识通信端口的句柄  
    DWORD dwEvtMask //能够使能的通信事件  
);
```

串口上可能发生的事件如下表所示：

值	事件描述
EV_BREAK	A break was detected on input.
EV_CTS	The CTS (clear-to-send) signal changed state.
EV_DSR	The DSR(data-set-ready) signal changed state.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state.
EV_RXCHAR	A character was received and placed in the input buffer.
EV_RXFLAG	The event character was received and placed in the input buffer. The event character is specified in the device's DCB structure, which is applied to a serial port by using the SetCommState function.
EV_TXEMPTY	The last character in the output buffer was sent.

在设置好事件掩模后，我们就可以利用 `WaitCommEvent()`函数来等待串口上发生事件，其函数原型为：

```
BOOL WaitCommEvent(  
    HANDLE hFile, //标识通信端口的句柄  
    LPDWORD lpEvtMask, //指向存放事件标识变量的指针  
    LPOVERLAPPED lpOverlapped, // 指向 overlapped 结构  
);
```

我们可以在发生事件后，根据相应的事件类型，进行串口的读写操作：

```
BOOL ReadFile(HANDLE hFile, //标识通信端口的句柄  
    LPVOID lpBuffer, //输入数据 Buffer 指针  
    DWORD nNumberOfBytesToRead, // 需要读取的字节数  
    LPDWORD lpNumberOfBytesRead, //实际读取的字节数指针  
    LPOVERLAPPED lpOverlapped //指向 overlapped 结构
```

```

);

BOOL WriteFile(HANDLE hFile, //标识通信端口的句柄
               LPCVOID lpBuffer, //输出数据 Buffer 指针
               DWORD nNumberOfBytesToWrite, //需要写的字节数
               LPDWORD lpNumberOfBytesWritten, //实际写入的字节数指针
               LPOVERLAPPED lpOverlapped //指向 overlapped 结构
);

```

2.工程实例

下面我们利用第 1 节所述 API 实现一个多线程的串口通信程序。这个例子工程（工程名为 MultiThreadCom）的界面很简单，如下图所示：



它是一个多线程的应用程序，包括两个工作者线程，分别处理串口 1 和串口 2。为了简化问题，我们让连接两个串口的电缆只包含 RX、TX 两根连线（即不以硬件控制 RS-232，串口上只会发生 EV_TXEMPTY、EV_RXCHAR 事件）。

在工程实例的 BOOL CMultiThreadComApp::InitInstance() 函数中，启动并设置 COM1 和 COM2，其源代码为：

```

BOOL CMultiThreadComApp::InitInstance()
{
    AfxEnableControlContainer();
    //打开并设置 COM1
    hComm1=CreateFile("COM1", GENERIC_READ|GENERIC_WRITE, 0,
                     NULL, OPEN_EXISTING, 0, NULL);
    if (hComm1==(HANDLE)-1)
    {
        AfxMessageBox("打开 COM1 失败");
        return false;
    }
}

```

```

        else
        {
            DCB wdcB;
            GetCommState (hComm1,&wdcB);
            wdcB.BaudRate=9600;
            SetCommState (hComm1,&wdcB);
            PurgeComm(hComm1,PURGE_TXCLEAR);
        }
        //打开并设置 COM2
        hComm2=CreateFile("COM2", GENERIC_READ|GENERIC_WRITE, 0,
            NULL ,OPEN_EXISTING, 0,NULL);
        if (hComm2==(HANDLE)-1)
        {
            AfxMessageBox("打开 COM2 失败");
            return false;
        }
        else
        {
            DCB wdcB;
            GetCommState (hComm2,&wdcB);
            wdcB.BaudRate=9600;
            SetCommState (hComm2,&wdcB);
            PurgeComm(hComm2,PURGE_TXCLEAR);
        }

        CMultiThreadComDlg dlg;
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal();
        if (nResponse == IDOK)
        {
            // TODO: Place code here to handle when the dialog is
            // dismissed with OK
        }
        else if (nResponse == IDCANCEL)
        {
            // TODO: Place code here to handle when the dialog is
            // dismissed with Cancel
        }
        return FALSE;
    }

```

此后我们在对话框 CMultiThreadComDlg 的初始化函数 OnInitDialog 中启动两个分别处理 COM1 和 COM2 的线程：

```

        BOOL CMultiThreadComDlg::OnInitDialog()
        {
            CDialog::OnInitDialog();

            // Add "About..." menu item to system menu.

            // IDM_ABOUTBOX must be in the system command range.
            ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
            ASSERT(IDM_ABOUTBOX < 0xF000);

            CMenu* pSysMenu = GetSystemMenu(FALSE);
            if (pSysMenu != NULL)
            {
                CString strAboutMenu;
                strAboutMenu.LoadString(IDS_ABOUTBOX);
                if (!strAboutMenu.IsEmpty())
                {
                    pSysMenu->AppendMenu(MF_SEPARATOR);
                    pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
                }
            }

            // Set the icon for this dialog. The framework does this automatically
            // when the application's main window is not a dialog
            SetIcon(m_hIcon, TRUE); // Set big icon
            SetIcon(m_hIcon, FALSE); // Set small icon

            // TODO: Add extra initialization here
            //启动串口 1 处理线程
            DWORD nThreadId1;

            hCommThread1 = ::CreateThread((LPSECURITY_ATTRIBUTES)NULL, 0,
            (LPTHREAD_START_ROUTINE)Com1ThreadProcess, AfxGetMainWnd()->m_hWnd, 0,
            &nThreadId1);
            if (hCommThread1 == NULL)
            {
                AfxMessageBox("创建串口 1 处理线程失败");
                return false;
            }
            //启动串口 2 处理线程
            DWORD nThreadId2;

            hCommThread2 = ::CreateThread((LPSECURITY_ATTRIBUTES)NULL, 0,
            (LPTHREAD_START_ROUTINE)Com2ThreadProcess, AfxGetMainWnd()->m_hWnd, 0,
            &nThreadId2);
            if (hCommThread2 == NULL)
            {

```

```

        AfxMessageBox("创建串口 2 处理线程失败");
        return false;
    }

    return TRUE; // return TRUE unless you set the focus to a control
}

```

两个串口 COM1 和 COM2 对应的线程处理函数等待串口上发生事件, 并根据事件类型和自身缓冲区是否有数据要发送进行相应的处理, 其源代码为:

```

DWORD WINAPI Com1ThreadProcess(HWND hWnd//主窗口句柄)
{
    DWORD wEven;
    char str[10]; //读入数据
    SetCommMask(hComm1, EV_RXCHAR | EV_TXEMPTY);
    while (TRUE)
    {
        WaitCommEvent(hComm1, &wEven, NULL);
        if(wEven == 0)
        {
            CloseHandle(hCommThread1);
            hCommThread1 = NULL;
            ExitThread(0);
        }
        else
        {
            switch (wEven)
            {
                case EV_TXEMPTY:
                    if (wTxPos < wTxLen)
                    {
                        //在串口 1 写入数据
                        DWORD wCount; //写入的字节数
                        WriteFile(hComm1, com1Data.TxBuf[wTxPos], 1, &wCount, NULL);
                        com1Data.wTxPos++;
                    }
                    break;
                case EV_RXCHAR:
                    if (com1Data.wRxPos < com1Data.wRxLen)
                    {
                        //读取串口数据, 处理收到的数据
                        DWORD wCount; //读取的字节数
                        ReadFile(hComm1, com1Data.RxBuf[wRxPos], 1, &wCount, NULL);
                        com1Data.wRxPos++;
                    }
                }
            }
        }
    }
}

```

```

        if(com1Data.wRxPos== com1Data.wRxLen);
        ::PostMessage(hWnd, COM_SENDCHAR, 0, 1);
    }
    break;
}
}
}
return TRUE;
}

```

DWORD WINAPI Com2ThreadProcess(HWND hWnd //主窗口句柄)

```

{
    DWORD wEven;
    char str[10]; //读入数据
    SetCommMask(hComm2, EV_RXCHAR | EV_TXEMPTY);
    while (TRUE)
    {
        WaitCommEvent(hComm2, &wEven, NULL);
        if (wEven = 0)
        {
            CloseHandle(hCommThread2);
            hCommThread2 = NULL;
            ExitThread(0);
        }
        else
        {
            switch (wEven)
            {
                case EV_TXEMPTY:
                    if (wTxPos < wTxLen)
                    {
                        //在串口 2 写入数据
                        DWORD wCount; //写入的字节数
                        WriteFile(hComm2, com2Data.TxBuf[wTxPos], 1, &wCount, NULL);
                        com2Data.wTxPos++;
                    }
                    break;
                case EV_RXCHAR:
                    if (com2Data.wRxPos < com2Data.wRxLen)
                    {
                        //读取串口数据, 处理收到的数据
                        DWORD wCount; //读取的字节数
                        ReadFile(hComm2, com2Data.RxBuf[wRxPos], 1, &wCount, NULL);

```



```

        com2Data.wRxPos++;
        if(com2Data.wRxPos== com2Data.wRxLen);
        ::PostMessage(hWnd, COM_SENDCHAR, 0, 1);
    }
    break;
}
}
}
return TRUE;
}

```

线程控制函数中所操作的 com1Data 和 com2Data 是与串口对应的数据结构 struct tagSerialPort 的实例，这个数据结构是：

```

typedef struct tagSerialPort
{
    BYTE RxBuf[SPRX_BUFLen]; //接收 Buffer
    WORD wRxPos; //当前接收字节位置
    WORD wRxLen; //要接收的字节数
    BYTE TxBuf[SPTX_BUFLen]; //发送 Buffer
    WORD wTxPos; //当前发送字节位置
    WORD wTxLen; //要发送的字节数
}SerialPort, * LPSerialPort;

```

3.多线程串口类

使用多线程串口通信更方便的途径是编写一个多线程的串口类，例如 Remon Spekreijse 编写了一个 CSerialPort 串口类。仔细分析这个类的源代码，将十分有助于我们对先前所学多线程及同步知识的理解。

3.1 类的定义

```

#ifndef __SERIALPORT_H__
#define __SERIALPORT_H__

#define WM_COMM_BREAK_DETECTED WM_USER+1 // A break was detected on input.
#define WM_COMM_CTS_DETECTED WM_USER+2 // The CTS (clear-to-send) signal changed
state.
#define WM_COMM_DSR_DETECTED WM_USER+3 // The DSR (data-set-ready) signal changed
state.
#define WM_COMM_ERR_DETECTED WM_USER+4 // A line-status error occurred. Line-status
errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
#define WM_COMM_RING_DETECTED WM_USER+5 // A ring indicator was detected.
#define WM_COMM_RLSD_DETECTED WM_USER+6 // The RLSD (receive-line-signal-detect)
signal changed state.

```

```

#define WM_COMM_RXCHAR WM_USER+7 // A character was received and placed in the input
                                buffer.
#define WM_COMM_RXFLAG_DETECTED WM_USER+8 // The event character was received and
                                placed in the input buffer.
#define WM_COMM_TXEMPTY_DETECTED WM_USER+9 // The last character in the output
                                buffer was sent.

class CSerialPort
{
public:
    // contruction and destruction
    CSerialPort();
    virtual ~CSerialPort();

    // port initialisation
    BOOL InitPort(CWnd* pPortOwner, UINT portnr = 1, UINT baud = 19200, char parity = 'N',
    UINT databits = 8, UINT stopbits = 1, DWORD dwCommEvents = EV_RXCHAR | EV_CTS, UINT
    nBufferSize = 512);

    // start/stop comm watching
    BOOL StartMonitoring();
    BOOL RestartMonitoring();
    BOOL StopMonitoring();

    DWORD GetWriteBufferSize();
    DWORD GetCommEvents();
    DCB GetDCB();

    void WriteToPort(char* string);

protected:
    // protected memberfunctions
    void ProcessErrorMessage(char* ErrorText);
    static UINT CommThread(LPVOID pParam);
    static void ReceiveChar(CSerialPort* port, COMSTAT comstat);
    static void WriteChar(CSerialPort* port);

    // thread
    CWinThread* m_Thread;

    // synchronisation objects
    CRITICAL_SECTION m_csCommunicationSync;
    BOOL m_bThreadAlive;

```

```

        // handles
HANDLE m_hShutdownEvent;
HANDLE m_hComm;
HANDLE m_hWriteEvent;

        // Event array.
        // One element is used for each event. There are two event handles for each port.
        // A Write event and a receive character event which is located in the overlapped structure
        (m_ov.hEvent).
        // There is a general shutdown when the port is closed.
HANDLE m_hEventArray[3];

        // structures
OVERLAPPED m_ov;
COMMTIMEOUTS m_CommTimeouts;
DCB m_dcb;

        // owner window
CWnd* m_pOwner;

        // misc
UINT m_nPortNr;
char* m_szWriteBuffer;
DWORD m_dwCommEvents;
DWORD m_nWriteBufferSize;
    };

#endif __SERIALPORT_H__

```

3.2 类的实现

3.2.1 构造函数与析构函数

进行相关变量的赋初值及内存恢复：

```

CSerialPort::CSerialPort()
{
    m_hComm = NULL;

    // initialize overlapped structure members to zero
    m_ov.Offset = 0;
    m_ov.OffsetHigh = 0;

    // create events
    m_ov.hEvent = NULL;
    m_hWriteEvent = NULL;
}

```

```

        m_hShutdownEvent = NULL;

        m_szWriteBuffer = NULL;

        m_bThreadAlive = FALSE;
    }

    //
    // Delete dynamic memory
    //
CSerialPort::~CSerialPort()
{
    do
    {
        SetEvent(m_hShutdownEvent);
    }
    while (m_bThreadAlive);

    TRACE("Thread ended\n");

    delete []m_szWriteBuffer;
}

```

3.2.2 核心函数：初始化串口

在初始化串口函数中，将打开串口，设置相关参数，并创建串口相关的用户控制事件，初始化临界区（Critical Section），以成队的 EnterCriticalSection()、LeaveCriticalSection()函数进行资源的排它性访问：

```

        BOOL CSerialPort::InitPort(CWnd *pPortOwner,
        // the owner (CWnd) of the port (receives message)
        UINT portnr, // portnumber (1..4)
        UINT baud, // baudrate
        char parity, // parity
        UINT databits, // databits
        UINT stopbits, // stopbits
        DWORD dwCommEvents, // EV_RXCHAR, EV_CTS etc
        UINT writebuffersize) // size to the writebuffer
    {
        assert(portnr > 0 && portnr < 5);
        assert(pPortOwner != NULL);

        // if the thread is alive: Kill
        if (m_bThreadAlive)
        {
            do

```

```

        {
            SetEvent(m_hShutdownEvent);
        }
        while (m_bThreadAlive);
        TRACE("Thread ended\n");
    }

    // create events
    if (m_ov.hEvent != NULL)
        ResetEvent(m_ov.hEvent);
    m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hWriteEvent != NULL)
        ResetEvent(m_hWriteEvent);
    m_hWriteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (m_hShutdownEvent != NULL)
        ResetEvent(m_hShutdownEvent);
    m_hShutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    // initialize the event objects
    m_hEventArray[0] = m_hShutdownEvent; // highest priority
    m_hEventArray[1] = m_ov.hEvent;
    m_hEventArray[2] = m_hWriteEvent;

    // initialize critical section
    InitializeCriticalSection(&m_csCommunicationSync);

    // set buffersize for writing and save the owner
    m_pOwner = pPortOwner;

    if (m_szWriteBuffer != NULL)
        delete []m_szWriteBuffer;
    m_szWriteBuffer = new char[writebuffersize];

    m_nPortNr = portnr;

    m_nWriteBufferSize = writebuffersize;
    m_dwCommEvents = dwCommEvents;

    BOOL bResult = FALSE;
    char *szPort = new char[50];
    char *szBaud = new char[50];

```

```

        // now it critical!
        EnterCriticalSection(&m_csCommunicationSync);

        // if the port is already opened: close it
        if (m_hComm != NULL)
        {
            CloseHandle(m_hComm);
            m_hComm = NULL;
        }

        // prepare port strings
        sprintf(szPort, "COM%d", portnr);
        sprintf(szBaud, "baud=%d parity=%c data=%d stop=%d", baud, parity, databits, stopbits);

        // get a handle to the port
        m_hComm = CreateFile(szPort, // communication port string (COMX)
            GENERIC_READ | GENERIC_WRITE, // read/write types
            0, // comm devices must be opened with exclusive access
            NULL, // no security attributes
            OPEN_EXISTING, // comm devices must use OPEN_EXISTING
            FILE_FLAG_OVERLAPPED, // Async I/O
            0); // template must be 0 for comm devices

        if (m_hComm == INVALID_HANDLE_VALUE)
        {
            // port not found
            delete []szPort;
            delete []szBaud;
            return FALSE;
        }

        // set the timeout values
        m_CommTimeouts.ReadIntervalTimeout = 1000;
        m_CommTimeouts.ReadTotalTimeoutMultiplier = 1000;
        m_CommTimeouts.ReadTotalTimeoutConstant = 1000;
        m_CommTimeouts.WriteTotalTimeoutMultiplier = 1000;
        m_CommTimeouts.WriteTotalTimeoutConstant = 1000;

        // configure
        if (SetCommTimeouts(m_hComm, &m_CommTimeouts))
        {
            if (SetCommMask(m_hComm, dwCommEvents))
            {
                if (GetCommState(m_hComm, &m_dcb))

```

```

        {
            m_dcb.fRtsControl = RTS_CONTROL_ENABLE; // set RTS bit high!
            if (BuildCommDCB(szBaud, &m_dcb))
            {
                if (SetCommState(m_hComm, &m_dcb))
                {
                    ;
                    // normal operation... continue
                }
                else
                {
                    ProcessErrorMessage("SetCommState()");
                }
            }
            else
            {
                ProcessErrorMessage("BuildCommDCB()");
            }
        }
        else
        {
            ProcessErrorMessage("GetCommState()");
        }
    }
    else
    {
        ProcessErrorMessage("SetCommMask()");
    }
}
else
{
    ProcessErrorMessage("SetCommTimeouts()");
}

delete []szPort;
delete []szBaud;

// flush the port
PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT |
    PURGE_TXABORT);

// release critical section
LeaveCriticalSection(&m_csCommunicationSync);

TRACE("Initialisation for communicationport %d completed.\nUse Startmonitor to
    communicate.\n", portnr);

return TRUE;
}

```

3.3.3 核心函数：串口线程控制函数

串口线程处理函数是整个类中最核心的部分，它主要完成两类工作：

- (1) 利用 `WaitCommEvent` 函数对串口上发生的事件进行获取并根据事件的不同类型进行相应的处理；
- (2) 利用 `WaitForMultipleObjects` 函数对串口相关的用户控制事件进行等待并做相应处理。

```

UINT CSerialPort::CommThread(LPVOID pParam)
{
    // Cast the void pointer passed to the thread back to
    // a pointer of CSerialPort class
    CSerialPort *port = (CSerialPort*)pParam;

    // Set the status variable in the dialog class to
    // TRUE to indicate the thread is running.
    port->m_bThreadAlive = TRUE;

    // Misc. variables
    DWORD BytesTransferred = 0;
    DWORD Event = 0;
    DWORD CommEvent = 0;
    DWORD dwError = 0;
    COMSTAT comstat;
    BOOL bResult = TRUE;

    // Clear comm buffers at startup
    if (port->m_hComm)
        // check if the port is opened
        PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR |
            PURGE_RXABORT | PURGE_TXABORT);

    // begin forever loop. This loop will run as long as the thread is alive.
    for (;;)
    {
        // Make a call to WaitCommEvent(). This call will return immediatly
        // because our port was created as an async port (FILE_FLAG_OVERLAPPED
        // and an m_OverlappedStructerlapped structure specified). This call will cause the
        // m_OverlappedStructerlapped element m_OverlappedStruct.hEvent, which is part of the
        // m_hEventArray to
        // be placed in a non-signeled state if there are no bytes available to be read,
        // or to a signeled state if there are bytes available. If this event handle
        // is set to the non-signeled state, it will be set to signeled when a
        // character arrives at the port.

        // we do this for each port!

        bResult = WaitCommEvent(port->m_hComm, &Event, &port->m_ov);

        if (!bResult)
        {
            // If WaitCommEvent() returns FALSE, process the last error to determin

```



```

        // the reason..
switch (dwError = GetLastError())
{
    case ERROR_IO_PENDING:
        {
            // This is a normal return value if there are no bytes
            // to read at the port.
            // Do nothing and continue
            break;
        }
    case 87:
        {
            // Under Windows NT, this value is returned for some reason.
            // I have not investigated why, but it is also a valid reply
            // Also do nothing and continue.
            break;
        }
    default:
        {
            // All other error codes indicate a serious error has
            // occurred. Process this error.
            port->ProcessErrorMessage("WaitCommEvent()");
            break;
        }
    }
else
{
    // If WaitCommEvent() returns TRUE, check to be sure there are
    // actually bytes in the buffer to read.
    //
    // If you are reading more than one byte at a time from the buffer
    // (which this program does not do) you will have the situation occur
    // where the first byte to arrive will cause the WaitForMultipleObjects()
    // function to stop waiting. The WaitForMultipleObjects() function
    // resets the event handle in m_OverlappedStruct.hEvent to the non-signelead state
    // as it returns.
    //
    // If in the time between the reset of this event and the call to
    // ReadFile() more bytes arrive, the m_OverlappedStruct.hEvent handle will be set again
    // to the signaled state. When the call to ReadFile() occurs, it will
    // read all of the bytes from the buffer, and the program will
    // loop back around to WaitCommEvent().
    //

```

```

// At this point you will be in the situation where m_OverlappedStruct.hEvent is set,
// but there are no bytes available to read. If you proceed and call
// ReadFile(), it will return immediatly due to the async port setup, but
// GetOverlappedResults() will not return until the next character arrives.
//
// It is not desirable for the GetOverlappedResults() function to be in
// this state. The thread shutdown event (event 0) and the WriteFile()
// event (Event2) will not work if the thread is blocked by GetOverlappedResults().
//
// The solution to this is to check the buffer with a call to ClearCommError().
// This call will reset the event handle, and if there are no bytes to read
// we can loop back through WaitCommEvent() again, then proceed.
// If there are really bytes to read, do nothing and proceed.

bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

if (comstat.cbInQue == 0)
    continue;
} // end if bResult

// Main wait function. This function will normally block the thread
// until one of nine events occur that require action.
Event = WaitForMultipleObjects(3, port->m_hEventArray, FALSE, INFINITE);

switch (Event)
{
case 0:
{
// Shutdown event. This is event zero so it will be
// the highest priority and be serviced first.

port->m_bThreadAlive = FALSE;

// Kill this thread. break is not needed, but makes me feel better.
AfxEndThread(100);
break;
}
case 1:
// read event
{
GetCommMask(port->m_hComm, &CommEvent);
if (CommEvent &EV_CTS)
::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_CTS_DETECTED,
(WPARAM)0, (LPARAM)port->m_nPortNr);
}
}

```

```

        if (CommEvent &EV_RXFLAG)
            ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_RXFLAG_DETECTED,(WPARAM)0, (LPARAM)port->m_nPortNr);
        if (CommEvent &EV_BREAK)
            ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_BREAK_DETECTED,(WPARAM)0, (LPARAM)port->m_nPortNr);
        if (CommEvent &EV_ERR)
            ::SendMessage(port->m_pOwner->m_hWnd, WM_COMM_ERR_DETECTED,
(WPARAM)0, (LPARAM)port->m_nPortNr);
        if (CommEvent &EV_RING)
            ::SendMessage(port->m_pOwner->m_hWnd,
WM_COMM_RING_DETECTED,(WPARAM)0, (LPARAM)port->m_nPortNr);
        if (CommEvent &EV_RXCHAR)
            // Receive character event from port.
            ReceiveChar(port, comstat);
            break;
        }
        case 2:
            // write event
            {
                // Write character event from port
                WriteChar(port);
                break;
            }
        } // end switch
    } // close forever loop
    return 0;
}

```

下列三个函数用于对串口线程进行启动、挂起和恢复：

```

//
// start comm watching
//
BOOL CSerialPort::StartMonitoring()
{
    if (!(m_Thread = AfxBeginThread(CommThread, this)))
        return FALSE;
    TRACE("Thread started\n");
    return TRUE;
}

//
// Restart the comm thread

```

```

//
BOOL CSerialPort::RestartMonitoring()
{
    TRACE("Thread resumed\n");
    m_Thread->ResumeThread();
    return TRUE;
}

//
// Suspend the comm thread
//
BOOL CSerialPort::StopMonitoring()
{
    TRACE("Thread suspended\n");
    m_Thread->SuspendThread();
    return TRUE;
}

```

3.3.4 读写串口

下面一组函数是用户对串口进行读写操作的接口：

```

//
// Write a character.
//
void CSerialPort::WriteChar(CSerialPort *port)
{
    BOOL bWrite = TRUE;
    BOOL bResult = TRUE;

    DWORD BytesSent = 0;

    ResetEvent(port->m_hWriteEvent);

    // Gain ownership of the critical section
    EnterCriticalSection(&port->m_csCommunicationSync);

    if (bWrite)
    {
        // Initailize variables
        port->m_ov.Offset = 0;
        port->m_ov.OffsetHigh = 0;

        // Clear buffer
        PurgeComm(port->m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR |

```

```
PURGE_RXABORT | PURGE_TXABORT);
```

```
bResult = WriteFile(port->m_hComm, // Handle to COMM Port  
port->m_szWriteBuffer, // Pointer to message buffer in calling function  
strlen((char*)port->m_szWriteBuffer), // Length of message to send  
&BytesSent, // Where to store the number of bytes sent  
&port->m_ov); // Overlapped structure
```

```
// deal with any error codes
```

```
if (!bResult)
```

```
{
```

```
DWORD dwError = GetLastError();
```

```
switch (dwError)
```

```
{
```

```
case ERROR_IO_PENDING:
```

```
{
```

```
// continue to GetOverlappedResults()
```

```
BytesSent = 0;
```

```
bWrite = FALSE;
```

```
break;
```

```
}
```

```
default:
```

```
{
```

```
// all other error codes
```

```
port->ProcessErrorMessage("WriteFile()");
```

```
}
```

```
}
```

```
}
```

```
else
```

```
{
```

```
LeaveCriticalSection(&port->m_csCommunicationSync);
```

```
}
```

```
} // end if(bWrite)
```

```
if (!bWrite)
```

```
{
```

```
bWrite = TRUE;
```

```
bResult = GetOverlappedResult(port->m_hComm, // Handle to COMM port
```

```
&port->m_ov, // Overlapped structure
```

```
&BytesSent, // Stores number of bytes sent
```

```
TRUE); // Wait flag
```

```
LeaveCriticalSection(&port->m_csCommunicationSync);
```

```

        // deal with the error code
        if (!bResult)
        {
            port->ProcessErrorMessage("GetOverlappedResults() in WriteFile()");
        }
    } // end if (!bWrite)

    // Verify that the data size send equals what we tried to send
    if (BytesSent != strlen((char*)port->m_szWriteBuffer))
    {
        TRACE("WARNING: WriteFile() error.. Bytes Sent: %d; Message Length: %d\n",
            BytesSent, strlen((char*)port->m_szWriteBuffer));
    }
}

//
// Character received. Inform the owner
//
void CSerialPort::ReceiveChar(CSerialPort *port, COMSTAT comstat)
{
    BOOL bRead = TRUE;
    BOOL bResult = TRUE;
    DWORD dwError = 0;
    DWORD BytesRead = 0;
    unsigned char RXBuff;

    for (;;)
    {
        // Gain ownership of the comm port critical section.
        // This process guarantees no other part of this program
        // is using the port object.

        EnterCriticalSection(&port->m_csCommunicationSync);

        // ClearCommError() will update the COMSTAT structure and
        // clear any other errors.

        bResult = ClearCommError(port->m_hComm, &dwError, &comstat);

        LeaveCriticalSection(&port->m_csCommunicationSync);

        // start forever loop. I use this type of loop because I
        // do not know at runtime how many loops this will have to

```

```

// run. My solution is to start a forever loop and to
// break out of it when I have processed all of the
// data available. Be careful with this approach and
// be sure your loop will exit.
// My reasons for this are not as clear in this sample
// as it is in my production code, but I have found this
// solution to be the most efficient way to do this.

```

```

        if (comstat.cbInQue == 0)
        {
            // break out when all bytes have been read
            break;
        }

EnterCriticalSection(&port->m_csCommunicationSync);

        if (bRead)
        {
bResult = ReadFile(port->m_hComm, // Handle to COMM port
                &RXBuff, // RX Buffer Pointer
                1, // Read one byte
                &BytesRead, // Stores number of bytes read
                &port->m_ov); // pointer to the m_ov structure
            // deal with the error code
            if (!bResult)
            {
                switch (dwError = GetLastError())
                {
                    case ERROR_IO_PENDING:
                    {
                        // asynchronous i/o is still in progress
                        // Proceed on to GetOverlappedResults();
                        bRead = FALSE;
                        break;
                    }
                    default:
                    {
                        // Another error has occurred. Process this error.
                        port->ProcessErrorMessage("ReadFile()");
                        break;
                    }
                }
            }
        }
        else

```

```

        {
// ReadFile() returned complete. It is not necessary to call GetOverlappedResults()
        bRead = TRUE;
        }
    } // close if (bRead)

    if (!bRead)
    {
        bRead = TRUE;
bResult = GetOverlappedResult(port->m_hComm, // Handle to COMM port
        &port->m_ov, // Overlapped structure
        &BytesRead, // Stores number of bytes read
        TRUE); // Wait flag

        // deal with the error code
        if (!bResult)
        {
port->ProcessErrorMessage("GetOverlappedResults() in ReadFile()");
        }
    } // close if (!bRead)

    LeaveCriticalSection(&port->m_csCommunicationSync);

    // notify parent that a byte was received
::SendMessage((port->m_pOwner)->m_hWnd, WM_COMM_RXCHAR,
    (WPARAM)RXBuff,(LPARAM)port->m_nPortNr);
    } // end forever loop

    }

    //

    // Write a string to the port
    //

void CSerialPort::WriteToPort(char *string)
    {
        assert(m_hComm != 0);

        memset(m_szWriteBuffer, 0, sizeof(m_szWriteBuffer));
        strcpy(m_szWriteBuffer, string);

        // set event for write
        SetEvent(m_hWriteEvent);
    }

```



```

//
// Return the output buffer size
//
DWORD CSerialPort::GetWriteBufferSize()
{
    return m_nWriteBufferSize;
}

```

3.3.5 控制接口

应用程序员使用下列一组 **public** 函数可以获取串口的 DCB 及串口上发生的事件：

```

//
// Return the device control block
//
DCB CSerialPort::GetDCB()
{
    return m_dcb;
}

//
// Return the communication event masks
//
DWORD CSerialPort::GetCommEvents()
{
    return m_dwCommEvents;
}

```

3.3.6 错误处理

```

//
// If there is a error, give the right message
//
void CSerialPort::ProcessErrorMessage(char *ErrorText)
{
    char *Temp = new char[200];

    LPVOID lpMsgBuf;

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, GetLastError(), MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        // Default language

```

```

(LPTSTR) &lpMsgBuf, 0, NULL);

sprintf(Temp,
"WARNING: %s Failed with the following error: \n%s\nPort: %d\n", (char*)
ErrorText, lpMsgBuf, m_nPortNr);
MessageBox(NULL, Temp, "Application Error", MB_ICONSTOP);

LocalFree(lpMsgBuf);
delete []Temp;
}

```

仔细分析 Remon Spekrijse 的 CSerialPort 类对我们理解多线程及其同步机制是大有益处的，从 <http://codeguru.earthweb.com/network/serialport.shtml> 我们可以获取 CSerialPort 类的介绍与工程实例。另外，电子工业出版社《Visual C++/Turbo C 串口通信编程实践》一书的作者龚建伟也编写了一个使用 CSerialPort 类的例子，可以从 <http://www.gjwtech.com/scomm/sc2serialportclass.htm> 获得详情。

4.多线程网络通信

在网络通信中使用多线程主要有两种途径，即主监控线程和线程池。

4.1 主监控线程

这种方式指的是程序中使用一个主线程监控某特定端口，一旦在这个端口上发生连接请求，则主监控线程动态使用 CreateThread 派生出新的子线程处理该请求。主线程在派生子线程后不再对子线程加以控制和调度，而由子线程独自和客户方发生连接并处理异常。

使用这种方法的优点是：

- (1) 可以较快地实现原型设计，尤其在用户数目较少、连接保持时间较长时有表现较好；
- (2) 主线程不与子线程发生通信，在一定程度上减少了系统资源的消耗。

其缺点是：

- (1) 生成和终止子线程的开销比较大；
- (2) 对远端用户的控制较弱。

这种多线程方式总的特点是"动态生成，静态调度"。

4.2 线程池

这种方式指的是主线程在初始化时静态地生成一定数量的悬挂子线程，放置于线程池中。随后，主线程将对这些悬挂子线程进行动态调度。一旦客户发出连接请求，主线程将从线程池中查找一个悬挂的子线程：

- (1) 如果找到，主线程将该连接分配给这个被发现的子线程。子线程从主线程处接管该连接，并与用户通信。当连接结束时，该子线程将自动悬挂，并进入线程池等待再次被调度；
- (2) 如果当前已没有可用的子线程，主线程将通告发起连接的客户。

使用这种方法进行设计的优点是：

- (1) 主线程可以更好地对派生的子线程进行控制和调度；
- (2) 对远程用户的监控和管理能力较强。

虽然主线程对子线程的调度要消耗一定的资源，但是与主监控线程方式中派生和终止线程所要耗费的资源相比，要少很多。因此，使用该种方法设计和实现的系统在客户端连接和终止变更频繁时有上佳表现。

这种多线程方式总的特点是"静态生成，动态调度"。