实验 2　windows 进程控制

（一）实验目的：

通过**创建进程**、**观察正在运行的进程**和**终止进程**的程序设计和调试操作，进一步熟悉操作系统的进程概念，理解 Windows 进程的“一生”，并了解学习创建进程、观察进程和终止进程的程序设计方法。

（二）实验内容：

在 windows 系统下通过有关进程的**系统调用**，实现进程的简单操作，并观察程序的运行情况，通过阅读和分析实验程序，分析执行结果，回答给出问题。这些加深对进程概念的理解，明确进程与程序之间的区别。

（三）实验要求：

(1)**理解系统调用 CreateProcess()的功能**。

(2)编译并运行给出的程序，独立观察和分析程序执行的结果，给出需要回答的问题。

(3)有多余时间则编写程序，实现输入任意执行文件并启动和选择结束进程功能的程序。

（四）实验步骤及具体内容：

1. 创建进程

本实验显示了创建子进程的基本框架。程序只是再一次地启动自身，并显示它的系统进程 ID 和它在进程列表中的位置。

清单 2-1　创建子进程

```
// proc create 项目
#include <windows.h>
#include <stdio.h>
#include <iostream.h>

// 创建传递过来的进程的克隆过程并赋于其 ID 值
void StartClone(int nCloneID)
{
    // 提取用于当前可执行文件的文件名
    TCHAR szFilename[MAX_PATH] ;
    GetModuleFileName(NULL, szFilename, MAX_PATH) ;

    // 格式化用于子进程的命令行并通知其 EXE 文件名和克隆 ID
    TCHAR szCmdLine[MAX_PATH];
     sprintf(szCmdLine,"\"%s\" %d",szFilename,nCloneID);

     // 用于子进程的 STARTUPINFO 结构
    STARTUPINFO si;
    ZeroMemory(reinterpret_cast <void*> (&si) , sizeof(si) ) ;
    si.cb = sizeof(si) ;                    // 必须是本结构的大小

    // 返回的用于子进程的进程信息
    PROCESS_INFORMATION pi;

    // 利用同样的可执行文件和命令行创建进程，并赋于其子进程的性质
    BOOL bCreateOK=CreateProcess(
        szFilename,                         // 新创建进程所使用的应用程序可执行文件的完整路径
        szCmdLine,                          // 新创建进程的命令行参数
```

```
            NULL,                   // 缺省的进程安全性
            NULL,                   // 缺省的线程安全性
            FALSE,                  // 不继承句柄
            CREATE_NEW_CONSOLE,     // 使用新的控制台
            NULL,                   // 新的环境
            NULL,                   // 当前目录
            &si,                    // 启动信息
            &pi) ;                  // 返回的进程信息

// 对子进程释放引用

    if (bCreateOK)
    {
        CloseHandle(pi.hProcess) ;
        CloseHandle(pi.hThread) ;
    }
}

int main(int argc, char* argv[] )
{
    // 确定进程在列表中的位置
    int nClone = 0 ;
    if (argc > 1)
    {
        sscanf(argv[1] , "%d" , &nClone) ;// 从第二个参数中提取克隆 ID
    }

    // 显示进程的 pid 值和 Clone 编号
    cout << "Process ID:" << GetCurrentProcessId()
                << ", Clone ID:" << nClone<< endl;

    const int c_nCloneMax=10;
     // 如果 nClone 的值小于 c_nCloneMax，则创建子进程
    if (nClone < c_nCloneMax)
    {
        StartClone(++nClone) ;// 发送新进程的命令行和克隆号
    }
    // 在终止之前暂停一下 (1/2 秒)
     Sleep(500) ;
     getchar(); //加上这一句可让进程停住（等待输入），方便观察。
    return 0;
}
```

步骤 1：编译并执行 2-1.exe 程序， 完成下列问题。

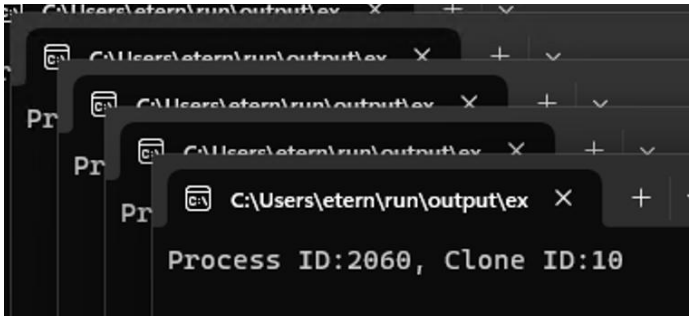
程序 2-1 是一个简单的使用 CreateProcess( ) API 函数的例子。首先形成简单的命令行，提供当前的 EXE 文件的指定文件名和代表生成克隆进程的号码。大多数参数都可取缺省值，但是**创建标志参数**使用了：

CREATE_NEW_CONSOLE

标志，**指示新进程分配它自己的控制台**，使得运行示例程序时，在任务栏上产生许多活动标记。然后该进程关闭传递过来的子进程的句柄并返回 main() 函数。在关闭程序之前，每一进程的执行主线程暂停一下，以便让用户看到其中的至少一个窗口。

程序运行时屏幕显示的信息是：

```
PS C:\Users\etern\run\output> & .\'exp1.1.exe'
Process ID:6684, Clone ID:0
```



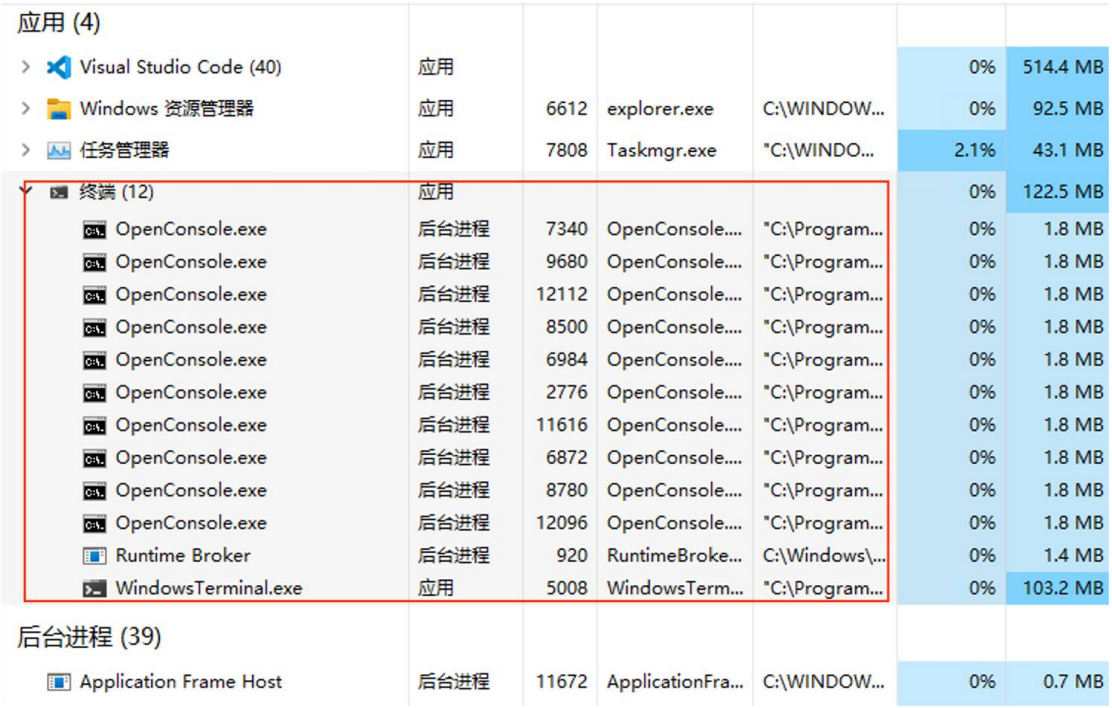
_____Process ID 和 Clone ID_____

步骤 2：单击 Ctrl + Alt + Del 键，进入“Windows 任务管理器”，在“应用程序”选项卡中，与程序运行前相比，有哪些变化？(**贴对比图，圈出来**)

应用 (3)						
Visual Studio Code (20)	应用				0%	501.0 MB
Windows 资源管理器	应用	6612	explorer.exe	C:\WINDOW...	0.3%	95.7 MB
任务管理器	应用	7808	Taskmgr.exe	"C:\WINDO...	0.9%	42.3 MB
应用 (4)						
Visual Studio Code (40)	应用				0%	514.2 MB
Windows 资源管理器	应用	6612	explorer.exe	C:\WINDOW...	0%	92.7 MB
任务管理器	应用	7808	Taskmgr.exe	"C:\WINDO...	4.2%	42.6 MB
终端 (12)	应用				0%	122.5 MB

步骤 3：在“Windows 任务管理器”的“进程”选项卡中，与程序运行前相比，有哪些变化？(**贴前后对比图，圈出来变换之处,用文字说明**)

应用 (3)						
Visual Studio Code (20)	应用				0%	487.7 MB
Windows 资源管理器	应用	6612	explorer.exe	C:\WINDOW...	0%	92.1 MB
任务管理器	应用	7808	Taskmgr.exe	"C:\WINDO...	2.3%	42.3 MB
后台进程 (37)						
Application Frame Host	后台进程	11672	ApplicationFra...	C:\WINDOW...	0%	0.8 MB
COM Surrogate	后台进程	7556	dllhost.exe	C:\WINDOW...	0%	0.1 MB
COM Surrogate	后台进程	6188	dllhost.exe	C:\WINDOW...	0%	0.4 MB
CTF 加载程序	后台进程	1512	ctfmon.exe	"ctfmon.exe"	0%	3.8 MB



应用 (4)						
Visual Studio Code (40)	应用				0%	514.4 MB
Windows 资源管理器	应用	6612	explorer.exe	C:\WINDOW...	0%	92.5 MB
任务管理器	应用	7808	Taskmgr.exe	"C:\WINDO...	2.1%	43.1 MB
终端 (12)	应用				0%	122.5 MB
OpenConsole.exe	后台进程	7340	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	9680	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	12112	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	8500	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	6984	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	2776	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	11616	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	6872	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	8780	OpenConsole....	"C:\Program...	0%	1.8 MB
OpenConsole.exe	后台进程	12096	OpenConsole....	"C:\Program...	0%	1.8 MB
Runtime Broker	后台进程	920	RuntimeBroke...	C:\Windows\...	0%	1.4 MB
WindowsTerminal.exe	应用	5008	WindowsTerm...	"C:\Program...	0%	103.2 MB
后台进程 (39)						
Application Frame Host	后台进程	11672	ApplicationFra...	C:\WINDOW...	0%	0.7 MB

有多个新的进程实例，它们的名称相同，PID 不同，命令行相同，占用内存相同。

为什么会产生这些变化?

每次执行 CreateProcess 函数时，都会创建一个新的进程副本。每个新进程都会在操作系统中注册，因此在任务管理器的“应用程序”和“进程”选项卡中都会显示这些新进程。

2. 进程的终止 **(注意理解：互斥锁是在哪儿创建的，怎使用的)**

在清单 2-2 列出的程序中，父进程中先创建一个互斥体 hMutexSuicide，在创建互斥体时通过设置第二个参数为 TRUE，从而占用了互斥体 hMutexSuicide，然后父进程通过系统调用 CreateProcess（）创建一个子进程，之后父进程主动睡眠（Sleep）5 秒；子进程创建后，打开父进程创建的互斥体 hMutexSuicide，然后通过系统调用“**WaitForSingleObject**(hMutexSuicide, INFINITE) ;”等待占用互斥体，由于互斥体当前被父进程占用，所以子进程将在此阻塞，一直阻塞到父进程通过系统调用“**ReleaseMutex**(hMutexSuicide) ;”释放了互斥体，子进程才能解除阻塞，继续运行，父进程和子进程各自独立运行完毕后，结束。

清单 2-2　子进程结束自己的父进程

```
// proc term 项目
# include <windows.h>
# include <stdio.h>
#include <iostream.h>
static LPCTSTR g_szMutexName = "w2kdg.ProcTerm.mutex.Suicide" ;

// 创建当前进程的克隆进程的简单方法
void StartClone(){
    // 提取当前可执行文件的文件名
    TCHAR szFilename[MAX_PATH] ;
    GetModuleFileName(NULL, szFilename, MAX_PATH) ;

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

    // 子进程的启动信息结构
    STARTUPINFO si;
```

```
    ZeroMemory(reinterpret_cast <void*>(&si),sizeof(si)) ;
    si.cb = sizeof(si) ;        // 应当是此结构的大小

    // 返回的用于子进程的进程信息
    PROCESS_INFORMATION pi;

    // 用同样的可执行文件名和命令行创建进程，并指明它是一个子进程
    BOOL bCreateOK=CreateProcess(
        szFilename,                 // 产生的应用程序的名称 (本 EXE 文件)
        szCmdLine,                  // 告诉我们这是一个子进程的标志
        NULL,                       // 用于进程的缺省的安全性
        NULL,                       // 用于线程的缺省安全性
        FALSE,                          // 不继承句柄
        CREATE_NEW_CONSOLE,         // 创建新窗口
        NULL,                       // 新环境
        NULL,                       // 当前目录
        &si,                        // 启动信息结构
        &pi ) ;                     // 返回的进程信息

    // 释放指向子进程的引用
    if (bCreateOK){
        CloseHandle(pi.hProcess) ;
        CloseHandle(pi.hThread) ;
    }
}

void Parent(){
    // 创建互斥体
    HANDLE hMutexSuicide= CreateMutex(
        NULL,                       // 缺省的安全性
        TRUE,                           // 最初拥有的
        g_szMutexName) ;            // 为其命名
    if (hMutexSuicide != NULL){
        // 创建子进程
        cout << "Creating the child process." << endl;
        StartClone() ;

        // 暂停
        Sleep(5000) ;

        // 释放占有的互斥体，从而使等待占有互斥体的子进程能够继续执行
        cout << "Telling the child process to quit. "<< endl;
        ReleaseMutex(hMutexSuicide) ;

        // 消除句柄
        CloseHandle(hMutexSuicide) ;
        getchar();
    }
}

void Child(){
    // 打开互斥体
    HANDLE hMutexSuicide = OpenMutex(
        SYNCHRONIZE,                // 打开用于同步
        FALSE,                          // 不需要向下传递
        g_szMutexName) ;            // 名称
    if (hMutexSuicide != NULL){
        // 报告我们正在等待指令
        cout <<"Child waiting for suicide instructions. " <<    endl;
        //因为互斥体 hMutexSuicide 当前被父进程占用，所以调用 WaitForSingleObject 子进程将阻塞在此，无法向下运行；直到父进程释放占有的互斥体后，才能解除阻塞，子进程继续向下执行
        WaitForSingleObject(hMutexSuicide, INFINITE) ;
        //子进程解除阻塞后，才能接着运行以下剩余的代码，全部运行完毕后，将退出。
```

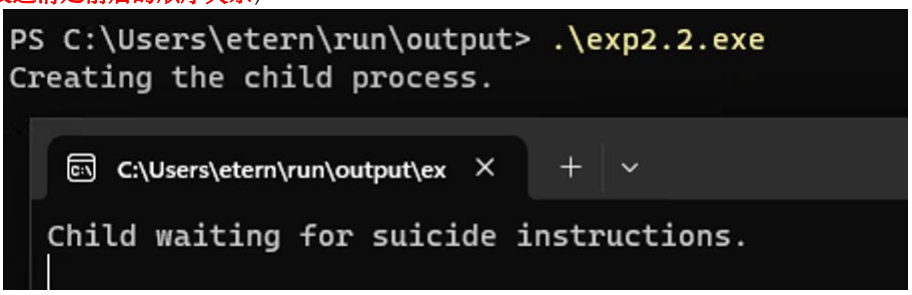
```
        // 准备好终止，清除句柄
        cout << "Child quiting." << endl;
        CloseHandle(hMutexSuicide) ;
        Sleep(5000) ;
        getchar();
    }
}

int main(int argc, char* argv[] ){
    // 决定其行为是父进程还是子进程
    if (argc > 1 && strcmp(argv[1] , "child" )== 0){
        Child() ;
    }
    else{
        Parent() ;
    }
    return 0;
}
```

清单 2-2 中的程序说明了一个进程从“生”到“死”的整个一生。第一次执行时，它创建一个子进程。在创建子进程之前，先创建一个互斥的内核对象，以便父子进程间进行协调。当创建子进程时，就打开了互斥体并在其线程中进行别的处理工作，然后调用 WaitForSingleObject()函数等待父进程释放互斥体信号，此时子进程将阻塞，等待着，直到父进程使用 ReleaseMutex() 发出信号，子进程将从阻塞中恢复，继续运行，并在退出前需要显式的关闭互斥体的句柄资源。

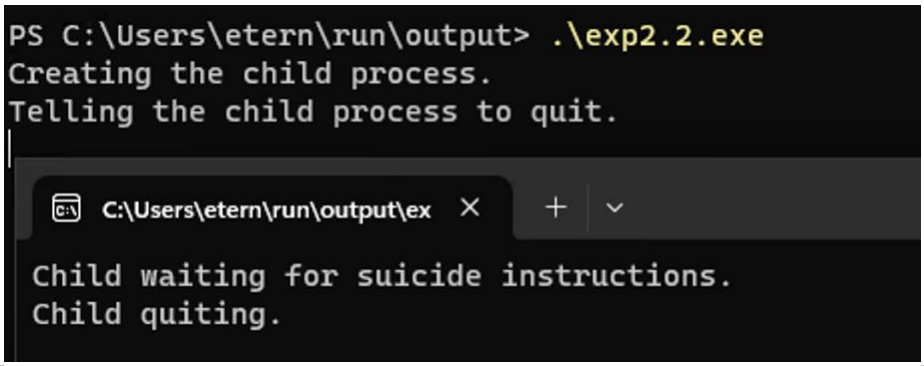
步骤 4：编译并执行 2-2.exe 程序， 其运行结果（**贴运行结果图，一个步骤一张图描述,标记上序号,表达清楚前后的顺序关系**）：

1)



表示：父进程创建互斥锁并占有，然后创建子进程并休眠 5 秒，子进程等待获取互斥锁

2)

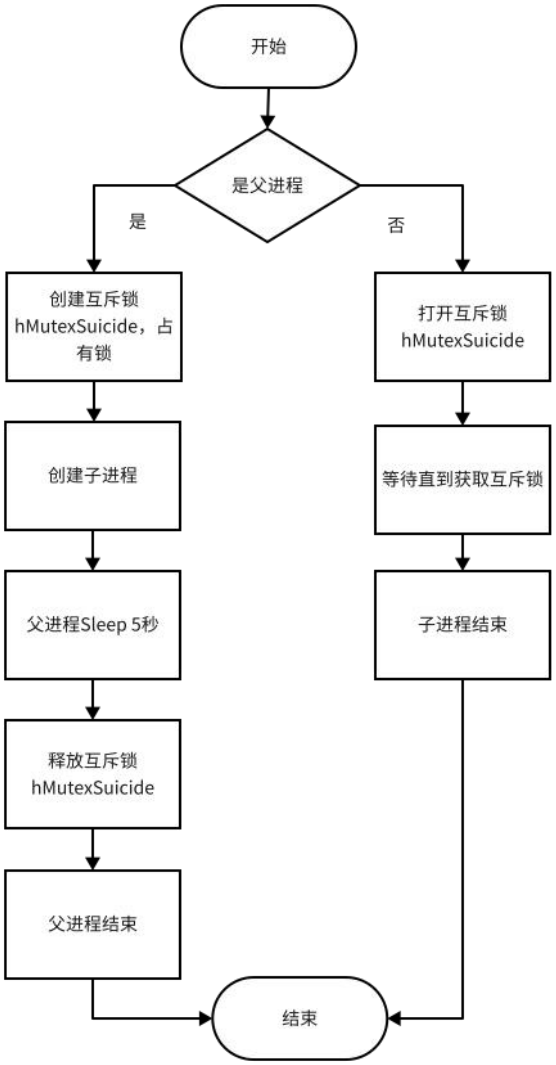


表示：__父进程苏醒并释放互斥锁，子进程获取获取互斥锁并继续执行

步骤 5：画流程图，用自己的语言解释该程序的运行过程：（**流程图,必须有“开始”和“结束”框，注意,不要出现环，即死循环**）

在运行时，检查它是作为父进程运行还是作为子进程运行。如果是父进程，创建一个名为 hMutexSuicide 的互斥锁，并立即占有它。然后，创建子进程并一直休眠 5 秒。在父进程休眠期间，子进程启动，尝试获取同一互斥锁。由于互斥锁已经被父进程占有，子进程在调用

WaitForSingleObject 时会阻塞。当父进程从休眠中醒来并释放互斥锁后，子进程就可以获取互斥锁并继续执行，最后关闭互斥锁的句柄并结束。



请用自己的语言说明该程序的运行方式与大家平日学习的 C 语言程序的运行方式有什么不同？

与平日学习的 C 语言程序相比，此程序使用了 Windows 特有的 API 调用和对象（比如互斥锁），并依赖于操作系统提供的并发执行和进程间同步的能力。该程序需要在程序逻辑中明确处理并发执行的情况和可能的竞态条件，这在单线程的 C 语言程序中通常不是必须考虑的。

（五）实验分析及小结

分析实验中用到的 API 函数的作用，系统的分析程序的运行过程和运行原理，系统的分析运行结果产生的原理。

实验分析：

CreateProcess() API 函数是 Windows 中用于创建新进程的关键函数。通过传入可执行文件路径、命令行参数、安全属性、继承选项、创建标志、环境变量、当前目录以及接收启动信息和进程信息的结构体，能够控制新进程的创建和行为。

在实验步骤 1 中，程序通过 CREATE_NEW_CONSOLE 标志创建了多个子进程，每个子进程都有自己的控制台窗口，并显示了各自的 Process ID 和 Clone ID，这验证了进程创建的实质是复制了一份当前执行的程序，并为其分配了独立的资源。

步骤 2 和步骤 3 中，通过观察任务管理器的变化，我可以直观地看到新进程的产生和区分。每个新创建的进程都有独立的 PID，但执行的程序（EXE 文件）和命令行参数是相同的。

实验步骤 4 和步骤 5 中的程序涉及到了进程同步的问题。互斥锁（Mutex）是一个同步机制，用于控制对共享资源的访问。在本实验中，互斥锁被用于父进程与子进程之间的同步。通过互斥锁，父进程能够控制子进程的执行时机，直到父进程释放互斥锁之后，子进程才继续执行。

小结：
本实验加深了我对 Windows 进程和进程控制的认识，特别是进程的创建、执行和终止。
我学习了如何使用 Windows API 进行进程控制，并理解了进程间同步的重要性和应用。
实验中的程序与传统的单线程 C 语言程序不同，它们涉及到了多进程并发执行和同步机制，这些是在并发编程中必须要考虑的问题。
本实验提升了我的问题解决能力，通过分析任务管理器中的变化，我能够从现象中推断出程序的运行状态和行为。
通过实验的执行和分析，我不仅掌握了进程控制的相关技术，也提升了我对操作系统中进程管理概念的理解。

附录-相关的背景知识：

Windows 所创建的每个进程都从系统调用 CreateProcess()函数开始，该函数的任务是在对象管理器子系统内初始化进程对象，而进程都以调用 ExitProcess() 或 TerminateProcess()函数终止。

1. 创建进程

CreateProcess() 调用的核心参数是可执行文件运行时的文件名及其命令行。下表详细地列出了每个参数的类型和名称。

CreateProcess() 函数的参数

参数名称	使用目的
LPCTSTR lpApplivationName	**指明包括可执行代码的 EXE 文件的文件名**
LPCTSTR lpCommandLine	**向可执行文件发送的参数**
LPSECURIITY_ATTRIBUTES lpProcessAttributes	返回进程的安全属性。
LPSECURIITY_ATTRIBUTES lpThreadAttributes	返回进程的主线程的安全属性
BOOL bInheritHandle	一种标志，告诉系统允许新进程继承创建者进程的句柄
DWORD dwCreationFlage	**特殊的创建标志（如 CREATE_SUSPENDED）的位标记**
LPVOID lpEnvironment	向新进程发送的一套环境变量；如为 null 值则发送调用者环境
LPCTSTR lpCurrentDirectory	新进程的启动目录

STARTUPINFO lpStartupInfo	STARTUPINFO 结构，包括新进程的输入和输出配置的信息
LPPROCESS_INFORMATION lpProcessInformation	应用程序的进程和主线程的句柄和 ID

可以指定第一个参数，即应用程序的名称，其中包括相对于当前进程的当前目录的全路径或者利用搜索方法找到的路径；lpCommandLine 参数允许调用者向新应用程序发送数据；接下来的三个参数与进程和它的主线程以及返回的指向该对象的句柄的安全性有关。

然后是**标志参数**，用以在 dwCreationFlags 参数中指明系统应该给予新进程什么行为。经常使用的标志是 CREATE_SUSPNDED，告诉主线程立刻暂停。当准备好时，应该使用 ResumeThread() API 来启动进程。**另一个常用的标志是 CREATE_NEW_CONSOLE**，告诉新进程启动自己的控制台窗口，而不是利用父窗口。这一参数还允许设置进程的优先级，用以向系统指明，相对于系统中所有其他的活动进程来说，给此进程多少 CPU 时间。

接着是 CreateProcess() 函数调用所需要的三个通常使用缺省值的参数。第一个参数是 lpEnvironment 参数，指明为新进程提供的环境；第二个参数是 lpCurrentDirectory，可用于向主创进程发送与缺省目录不同的新进程使用的特殊的当前目录；第三个参数是 STARTUPINFO 数据结构所必需的，用于在必要时指明新应用程序的主窗口的外观。

CreateProcess() 的最后一个参数是用于新进程对象及其主线程的句柄和 ID 的返回值缓冲区。调用 CloseHandle() API 函数关闭 PROCESS_INFORMATION 结构中返回的句柄，是重要的，因为如果不将这些句柄关闭的话，当子进程终止后，系统仍然不能完全清理与子进程相关的数据结构，因为父进程依然有未关闭的子进程的句柄。

参考：http://msdn.microsoft.com/en-us/library/ms682425(v=vs.85).aspx

参考：http://baike.baidu.com/view/697167.htm

参考：http://msdn.microsoft.com/en-us/library/ms684873(v=vs.85).aspx

PROCESS_INFORMATION Structure

- Syntax

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

- Members

hProcess

A handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread

A handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId

A value that can be used to identify a process. The value is valid from the time the process is created until all handles to the process are closed and the process object is freed; at this point, the identifier may be reused.

dwThreadId

A value that can be used to identify a thread. The value is valid from the time the thread is created until all handles to the thread are closed and the thread object is freed; at this point, the identifier may be reused.

- Remarks

If the function succeeds, be sure to call the CloseHandle function to close the hProcess and hThread handles when you are finished with them. Otherwise, when the child process exits, the

system cannot clean up the process structures for the child process because the parent process still has open handles to the child process. However, the system will close these handles when the parent process terminates, so the structures related to the child process object would be cleaned up at this point.

2. 正在运行的进程

如果一个进程拥有至少一个执行线程，则为正在系统中运行的进程。当调用 ExitProcess() API 函数，通知系统终止它所拥有的所有正在运行、准备运行或正在挂起的其他线程。当进程正在运行时，可以查看它的许多特性。

首先可利用 GetCurrentProcessId() 函数来查看的进程特性是进程标识符 (PID)，返回的 PID 在整个系统中都可使用。其他的可显示当前进程信息的 API 函数如 GetStartupInfo()和 GetProcessShutdownParameters() 可给出进程的配置信息。

通常，一个进程需要它的运行期环境的信息。例如 API 函数 GetModuleFileName() 和 GetCommandLine()，可以给出用在 CreateProcess() 中的参数以启动应用程序。

用 GetGuiResources() 来查看进程的 GUI 资源。此函数既可返回指定进程中的打开的 GUI 对象的数目，也可返回指定进程中打开的 USER 对象的数目。进程的其他性能信息可通过 GetProcessIoCounters()、GetProcessPriorityBoost()、GetProcessTimes()、GetProcessWorkingSetSize() 和 GetProcessVersion()得到。

3. 终止进程

所有进程都是以调用 ExitProcess() 或者 TerminateProcess() 函数结束的。最好用前者而不要用后者，因为进程是在完成了它的所有的关闭任务之后以正常的终止方式来调用前者的。而外部进程通常调用后者即突然终止进程的进行，由于关闭时的途径不太正常，有可能引起错误的行为。

操作系统	版本号
Windows 7	6.1
Windows Server 2008 R2	6.1
Windows Server 2008	6.0
Windows Vista	6.0
Windows Server 2003 R2	5.2
Windows Server 2003	5.2
Windows XP	5.1
Windows 2000	5.0

4. 互斥量（也成为互斥体、Mutex）

互斥量包含的几个操作原语：

http://msdn.microsoft.com/en-us/library/ms684266(VS.85).aspx

- ✓ 创建一个互斥量： CreateMutex（）
- ✓ 打开一个互斥量 ： OpenMutex（）

✓ 释放互斥量: ReleaseMutex（）
✓ 等待互斥量对象: WaitForSingleObject（）和 WaitForMultipleObjects（）

（1）创建一个互斥量 CreateMutex（）**函数原型**:

```
HANDLE  CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes, //Mutex 的安全属性，可为 NULL
  BOOL  bInitialOwner,    //TRUE 表示创建者立即占用该 Mutex；FALSE 表示不占用
  LPCTSTR  lpName       //Mutex 的名字，需保证该名字在系统中唯一，用于跨进程访问
  );
```

返回值:

如果函数调用成功，返回值是互斥对象句柄；如果函数调用之前，有名互斥对象已存在，那么函数给已存在的对象返回一个句柄，并且函数 GetLastError 返回 ERROR_ALREADY_EXISTS，否则，调用者创建互斥对象。

如果函数调用失败，则返回值为 NULL。若想获得更多错误信息，请调用 GetLastError 函数。

（2）打开一个互斥量 OpenMutex（）**函数原型**:

```
HANDLE  OpenMutex(
  DWORD dwDesiredAccess, //希望获得的 Mutex 的访问属性,
                         //例如 SYNCHRONIZE 或 MUTEX_ALL_ACCESS
  BOOL  bInheritHandle,    //TRUE 表示子进程可以继承该 Mutex 的句柄; FALSE 表示子
                           进程不可继承该 Mutex 的句柄
  LPCTSTR  lpName       //想打开的 Mutex 的名字，需保证该名字所对应 Mutex 已经创建
  );
```

返回值:

如果函数调用成功，返回希望打开的 Mutex 的句柄；失败，返回 NULL。

同步对象的访问属性可以参考:
http://msdn.microsoft.com/en-us/library/ms686670(v=VS.85).aspx

（3）释放互斥量 ReleaseMutex **函数原型**:

```
BOOL  ReleaseMutex( HANDLE hMutex ) ; //释放由参数 hMutex 指向的 Mutex，相当于
                                        signal(Mutex)操作
```

（4）等待互斥量对象 **WaitForSingleObject 函数原型**

```
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
);
```

附录 2　扩展训练

该部分为扩展训练，供有兴趣和时间的同学参考，不做要求。

综合利用前面的知识了解一个简单的“看门狗”的实现原理，该内容来自网页 http://blog.csdn.net/diebird/article/details/303821

给应用程序加装“看门狗”

相信大多数的程序员或用户，在 Windows 中见到类似于下面的亲切而又温馨的提示信息，都不会感到陌生：
“XXX 执行了非法操作，将被关闭。要终止程序，请单击<确定>；要调试程序，请单击<取消>。”或者，“是否向 Microsoft 发送错误报告？<发送>，<不发送>。”

如果这个程序运行在无人值守、需要保持连续工作状态的场合，而其中的 bug 又一时难以排除，就需要采取应急措施，消除或减少程序出错造成的影响。本文讨论解决这个问题的办法。

做过一定硬件开发的人都知道，恶劣的工作环境，带有缺陷的硬件设计，不完善的算法等内外因素，都可能造成程序“跑飞”，因此专门加装一个“看门狗”，负责监视程序主体，必要时产生复位中断，有效地避免设备当机。

“看门狗”的思想，完全可以拿到高级语言编程中来用。基本做法是：设计一个简单的监视程序做为主进程，将原来的工作程序作为子进程，由主进程启动子进程并监视子进程的运行状态。子进程在发生严重错误时不弹出本文开始时描述的对话框，而是悄悄退出。主进程发现子进程退出后，重新启动子进程。如此反复。

在具体实现上，下面以 VC 为例说明：

设置子进程为“静默模式”

在系统初始化部分(CWinApp 或 main 中的开头)，调用 API 函数 SetErrorMode
SetErrorMode(SEM_NOGPFAULTERRORBOX);
保证程序在发生严重错误时不弹出对话框，无需人工干预，自行退出。

启动子进程

在主进程中，创建子进程并运行。假定子进程的可执行文件为 work.exe，示意性代码如下

STARTUPINFO si; PROCESS_INFORMATION pi; ZeroMemory(&pi, sizeof(pi)); ZeroMemory(&si, sizeof(si)); si.cb = sizeof(si); // Start the child process if (CreateProcess("work.exe", "", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) { // success … … }
CreateProcess 有 10 个参数，看起来挺吓人，其实并不复杂，很容易理解。最后一个参数会返回子进程的 ID 和句柄等信息，后面就是对进程 ID 或句柄进行监视。

监视子进程

定时检查子进程是否在正常运行。有好几个 API 都可以用于对指定 ID 的进程进行监视，象 GetProcessVersion，GetProcessTimes，GetProcessIoCounters 等，其中 GetProcessVersion 最简单，只有一个参数：

DWORD GetProcessVersion(DWORD ProcessId);
当子进程已经退出时，该函数返回 0。

更为“专业”的函数是 GetExitCodeProcess，它甚至能告诉我们子进程退出的原因:

BOOL GetExitCodeProcess(HANDLE hProcess, // handle to the process LPDWORD lpExitCode // termination status);

更进一步的考虑

为增强系统的可靠性，给工作程序加装“看门狗”，不失为一种可行的技术方案。但如果有两套用户界面，看起来就有点不那么专业了。可将子进程设计为基于 console 的应用，不带用户界面，所有的信息都通过主进程窗口输出。主进程 CreateProcess 的第 6 个参数需加入 CREATE_NO_WINDOW 项，将子进程隐藏起来。这样从用户的角度看起来，就象只存在一个应用程序。

另一个问题是，如果用户关闭主进程，如何同时关闭子进程？用 TerminateProcess 函数固然能结束子进程，但可能会造成内存泄漏等新问题。最好是主进程向子进程发出结束的消息并进行同步，使子进程能够从容地退出。

再扩展一下，一个主进程可以同时管理多个子进程。典型的例子是利用多块网卡进行抓包、分析、处理的系统，将每一块网卡应用与一个子进程绑定，而主进程负责监视所有的子进程的工作。

上面的讨论涉及到进程间通信(IPC)问题。解决的办法有很多，象 file mapping, mailslot, pipe, DDE, COM, RPC, clipboard, socket, WM_COPYDATA 等都能达到目的，可根据个人喜好和具体情况采用。