Windows 2000 应用系列

Windows 2000 编程

李多多 等　编著

人 民 邮 电 出 版 社

第 9 章　进程与线程

本章将全面接触系统编程中的核心问题：进程与线程。

9.1 多任务

9.1.1　多任务概述

假如你有一个克隆的话，你就能同时做两件事。对于有多个 CPU 的计算机，同时在每一个 CPU 上运行程序称为多重处理。但是，你不能在没有多个处理器的计算机上进行多重处理。因此，如果你正在使用一台只有一个处理器的计算机，操作系统可以进行多任务处理，如果你正在使用一台有多个处理器的计算机，操作系统既能进行多任务处理又能进行多重处理。

在操作系统中，进程是一个逻辑上的任务。进程是在运行应用程序、启动某一系统服务和在 Windows NT 中启动某一子系统时产生的。每一个进程都有自己的专用资源，只有拥有这个进程的应用程序才能访问这些资源。这意味着，如果你产生了一个程序，在程序中用到一些数据，并且你也没有建立任何形式的程序间数据共享机制或使用操作系统的程序间数据共享机制，那么就只有你的程序能访问这些数据。

多任务处理有两种基本方法：

- 协同多任务。在这种方法中，正在运行的进程必须为其他进程留出 CPU 时间片。即每个程序必须允许其他程序使用 CPU，它们都有一个特殊的码环，这个码环产生控制允许其他应用软件的运行。
- 抢先式多任务。在这种方法中，操作系统决定哪个程序获得时间片，一个应用软件在任何时候都有可能被操作系统暂停。

Microsoft Windows 3.x 和 Macintosh 用的都是协同多任务处理，而 OS/2、Windows 95/98/NT/2000、UNIX 操作系统使用的是抢先式多任务处理。Windows 95 对于 32 位 Windows 应用程序采用抢先式多任务处理，为了能够向下兼容，对于 16 位的 Windows 应用程序，即为 Windows 3.x 写的应用程序仍采用协同多任务处理。

9.1.2　协同式多任务

在 16 位的 Windows 3.x 中，应用程序具有对 CPU 的控制权。只有在调用了 GetMessage、PeekMessage、WaitMessage 或 Yield 后，程序才有可能把 CPU 控制权交给系统，系统再把控制权转交给别的应用程序。如果应用程序在长时间内无法调用上述 4 个函数之一，那么程序就一直独占 CPU，系统会被挂起而无法接受用户的输入。

因此，在设计 16 位的应用程序时，程序员必须合理地设计消息处理函数，以使程序能够尽快返回到消息循环中。如果程序需要进行费时的操作，那么必须保证程序在进行操作时能周期性地调用上述 4 个函数中的一个。

在 Windows 3.x 环境下，要想设计一个既能执行实时的后台工作（如对通信端口的实时监测和读写），又能保证所有界面响应用户输入的单独的应用程序几乎是不可能的。

有人可能会想到用 CWinApp::OnIdle 函数来执行后台工作，因为该函数是程序主消息循环在空闲时调用的。但 OnIdle 的执行并不可靠，例如，如果用户在程序中打开了一个菜单或模态对话框，那么 OnIdle 将停止调用，因为此时程序不能返回到主消息循环中。在实时任务代码中调用 PeekMessage 也会遇到同样的问题，除非程序能保证用户不会选择菜单或弹出模态对话框，否则程序将不能返回到 PeekMessage 的调用处，这将导致后台实时处理的中断。

折衷的办法是在执行长期工作时弹出一个非模态对话框并禁止主窗口，在消息循环内分批执行后台操作。对话框中可以显示工作的进度，也可以包含一个取消按钮以让用户有机会中断一个长期的工作。这样做既可以保证工作实时进行，又可以使程序能有限地响应用户输入，但此时程序实际上已不能再为用户干别的事情了。典型的代码如下所示。

```
//协同多任务环境下防止程序被挂起的一种方法
bAbort=FALSE;
lpMyDlgProc=MakeProcInstance(MyDlgProc, hInst);
hMyDlg=CreateDialog (hInst, "Abort", hwnd, lpMyDlgProc); //创建一个非模态对话框
ShowWindow(hMyDlg, SW_NORMAL);
UpdateWindow(hMyDlg);
EnableWindow(hwnd, FALSE); //禁止主窗口
. . .
while(!bAbort)
{
. . . //执行一次后台操作
. . .
while(PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
{
if(!IsDialogMessage(hMyDlg, &msg))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
```


此为试读，需要完整PDF请访问：www.ertongbook.com

```
}
}
}
EnableWindow(hwnd, TRUE); //允许主窗口
DestroyWindow(hMyDlg);
FreeProcInstance(lpMyDlgProc);
```

9.1.3 抢先式多任务

在 32 位的 Windows 系统中，采用的是抢先式多任务，这意味着程序对 CPU 的占用时间是由系统决定的。系统为每个程序分配一定的 CPU 时间，当程序的运行超过规定时间后，系统就会中断该程序并把 CPU 控制权转交给别的程序。与协同式多任务不同，这种中断是汇编语言级的。程序不必调用像 PeekMessage 这样的函数来放弃对 CPU 的控制权，就可以进行费时的工作，而且不会导致系统的挂起。

在 Windows3.x 中，如果某一个应用程序陷入了死循环，那么整个系统都会瘫痪，这时唯一的解决办法就是重新启动机器。而在 Windows 95/98/2000/NT 中，一个程序的崩溃一般不会造成死机，其他程序仍然可以运行，用户可以按 Ctrl+Alt+Del 键来打开任务列表并关闭没有响应的程序。

9.2 进程

9.2.1 进程概述

在 Window 系统中，进程一般被看成程序执行的一个实例。进程之间一般是相互独立的，每个进程独占系统所分配的地址控件，而不受其他进程的影响。当然，当你用钩子函数或远程线程打破其他进程的边界墙时，他们之间就不再是完全独立的了。

在 Win32 下，进程是没有活力的，也就是说，进程不做实际的工作，它只占用 4GB 的虚拟地址空间。在此空间内，有应用程序 EXE 所需要的代码和数据。EXE 文件所需要的动态链接库的代码段和数据段也加载入此空间。应用程序所需的所有资源、文件、动态内存分配，以及进程所创建的线程也都载入此段。在进程终止时，所有这些资源都将由操作系统自动释放。

为了让进程完成实际的工作，必须在进程中创建线程。线程负责执行包含装载进程地址空间内的所有代码。一个进程可以创建多个线程，而一个线程又可以创建多个子线程。当一个进程中的所有线程都结束时，操作系统会自动结束该进程。进程与线程的关系如图 9-1 所示。

多线程的运行机制来源于并行处理。Win32 允许 CPU 在多个进程之间切换，以便使用户看起来好像系统在同时处理多个任务。实际上，Win32 所采用的是任务切换模式。即系统在多个进程之间按照一定的规则切分时间片。不言而喻，单个 CPU 在这种工作模式下，实际上会降低系统的执行效率。由于 Windows 2000 继承了相当一部分 Window NT 的特性，

支持多个 CPU 同时工作,因而在对多线程的处理上表现了较好的性能。这就意味着 Windows 2000 可以在多处理器计算机上同时处理多个线程。

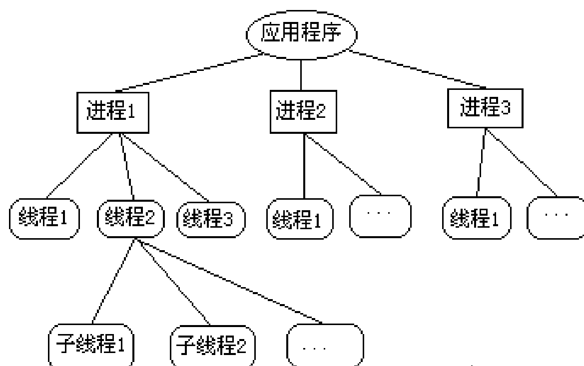

图 9-1　程序、进程与线程间的关系

9.2.2　WinMain 函数初探

WinMain 函数是 Windows 2000 应用程序的入口函数,读者可能很少关注过其中的参数,下面让我们来看一下。函数原型如下:

```
int WINAPI WinMain(  HINSTANCE hInstance,      // 当前进程实例句柄
                     HINSTANCE hPrevInstance,  // 先前进程实际句柄
                     LPSTR lpCmdLine,          // 命令行参数
                       int nCmdShow            //应用程序窗口样式);
```

hInstance:当前进程实例句柄。在 Win32 中，每个装入内存中的 EXE 或 DLL 都有一个实例句柄。实际上，进程实例句柄为装入内存的 EXE（或 DLL）的内存基地址。Win32 将从这个基地址开始查找所需要的各种资源。熟悉汇编语言的程序开发人员不难推想出这种资源在内存中的存储方式。当然，作为应用程序开发人员，不需要了解太多底层事务，操作系统与 API 层会处理这一切。

hPrevInstance　先前进程实际句柄。Win32 中此参数毫无用处。通常此值为 NULL。

lpCmdLine　命令行参数。有经验的程序员一眼就可以注意到，此参数为一 ANSI 字符集。这是出于在 Windows 发展进程中的兼容性原因而形成的。自然，这将对应用程序中 Unicode 的使用造成麻烦。出于这个原因，Microsoft 提供以下一个 API 作为补充:LPTSTR GetCommandLine(VOID);这个 API 直接返回 Unicode 字符串，以确保 ANSI 与 Unicode 同时被处理。

nCmdShow　:　应用程序创建窗口的样式。常用的窗口样式如表 9-1 所示。

进程的实例句柄有两种用途:

(1) 从 EXE 或 DLL 中加载资源。这些资源包括:位图、图标、菜单、声音文件，以及 DLL 中特有的函数模块。许多应用程序将所用的资源打包到 EXE 或 DLL 中，以减少程序执行时所需文件的个数。自然，这会引起 EXE 或 DLL 体积巨大。当程序需要使用这些资源

时，将用到如下函数：

表 9-1 窗口样式

窗口样式	标记意义
SW_HIDE	隐藏窗口
SW_MINIMIZE	最小化窗口
SW_SHOW	在当前位置显示普通窗口
SW_SHOWMAXIMIZED	激活窗口并最大化
SW_SHOWMINIMIZED	激活窗口并最小化为图标
SW_SHOWNORMAL	将最小化窗口或最大化窗口恢复原状

- 从资源中加载光标：

```
HICON LoadIcon(  HINSTANCE hInstance, //  当前进程实例句柄
                 LPCTSTR  lpIconName     //光标文件名 );
```

- 从资源中加载图标：

```
HANDLE LoadImage(
   HINSTANCE  hinst,     //  当前进程实例句柄
   LPCTSTR  lpszName,  //  图标文件名
   UINT  uType,
   int  cyDesired,
   UINT  fuLoad            );
```

- 从资源中加载位图：

```
HBITMAP LoadBitmap(  HINSTANCE  hInstance,   //当前进程实例句柄
                     LPCTSTR  lpBitmapName   //  位图文件名);
```

(2) 操作进程所创建的线程。

如果不在 WinMain 中获取进程实例句柄，可采用如下 API：

```
HMODULE  GetModuleHandle(   LPCTSTR  lpModuleName     );
```

lpModuleName：EXE 或 DLL 的文件名。

此返回值即为进程实例句柄。在 Win32 中，HMODULE 与 HINSTANCE 可以不加区别地引用。

9.2.3 创建进程

1．进程创建函数详解

Win32 提供如下 API 函数创建进程：

```
BOOL CreateProcess(
   LPCTSTR lpApplicationName,                          //  可执行文件名
   LPTSTR  lpCommandLine,                              //  命令行参数缓冲区指针
   LPSECURITY_ATTRIBUTES  lpProcessAttributes,      //  进程安全属性
```

```
LPSECURITY_ATTRIBUTES lpThreadAttributes,   // 线程安全属性
BOOL bInheritHandles,             // 是否从父进程继承
DWORD dwCreationFlags,            // 进程创建标志
LPVOID lpEnvironment,             // 新环境块指针
LPCTSTR lpCurrentDirectory,       // 进程当前目录
LPSTARTUPINFO lpStartupInfo, // STARTUPINFO 指针
LPPROCESS_INFORMATION
        lpProcessInformation // PROCESS_INFORMATION 指针
);
```

参数说明如下：

lpApplicationName　**当前进程所创建的子进程的可执行文件名。可以为文件系统全路径名，也可以为相对路径名。**

lpCommandLine　**新进程的命令行参数。**

lpProcessAttributes **和** lpThreadAttributes　**进程、线程安全属性结构。该结构为新进程和它所创建的线程设置安全属性。安全属性结构包括如下内容：**

```
typedef struct _SECURITY_ATTRIBUTES { // sa
DWORD   nLength;  //按字节计算的结构长度
LPVOID lpSecurityDescriptor; //安全信息数据指针
BOOL    bInheritHandle;      //是否从父进程继承句柄
} SECURITY_ATTRIBUTES;
```

此参数可以为 NULL**，此时新进程将从创建进程继承安全属性。**

表 9-2　　**进程的产生方式**

创建标记	进程的创建方式
CREATE_DEFAULT_ERROR_MODE	新进程不继承创建进程的出错处理模式。CreateProcess 将赋予此进程当前默认的出错处理方式
CREATE_NEW_CONSOLE	新进程拥有一个新的控制台，而并非从父进程继承。此标记不可与 DETACHED_PROCESS 同时使用
CREATE_SUSPENDED	新进程的主线程以挂起的状态被启动，直到 ResumeThread 被调用为止
CREATE_NEW_PROCESS_GROUP	新进程是一组新进程的根进程
CREATE_SEPARATE_WOW_VDM	仅对 19-bit Window 应用程序有效。新进程将在独立的虚拟 DOS 机(VDM)上运行
CREATE_SHARED_WOW_VDM	仅对 19-bit Window 应用程序有效。新进程将在共享的虚拟 DOS 机(VDM)上运行
CREATE_UNICODE_ENVIRONMENT	进程环境变量使用 Unicode 字符集。默认时，将使用 ANSI 字符集
HIGH_PRIORITY_CLASS	为进程设置高优先级级别。此进程必须立即被执行。过多的高优先级进程会影响系统的执行效率。例如：WindowsNT 任务管理器
IDLE_PRIORITY_CLASS	空闲优先级。此进程的线程只在系统空闲时执行。例如：屏幕保护程序
NORMAL_PRIORITY_CLASS	普通优先级
REALTIME_PRIORITY_CLASS REALTIME_PRIORITY_CLASS	实时优先级，为最高优先级。可能高于后台运行的系统级进程。实时优先级使用不当，会使系统某些操作无反映

此为试读，需要完整PDF请访问：www.ertongbook.com

dwCreationFlags 新进程的产生方式。几乎所有这些标志都可以用按位或组合来连接，除了涉及到进程优先级的标记以外。可以参照如表 9-2 设置进程方式。

lpCurrentDirectory 新进程的当前目录。如果为 NULL，将同创建它的进程拥有相同当前目录。

lpStartupInfo 指向 STARTUPINFO 结构的指针。此结构包含应用程序开始运行时主窗口的状态。此结构的具体信息为：

```
typedef struct _STARTUPINFO { // si
    DWORD    cb;            //以字节为单位的结构大小
    LPTSTR   lpReserved; //保留待用，必须为 NULL
    LPTSTR   lpDesktop; //应用程序启动的桌面
    LPTSTR   lpTitle;       //窗口标题
    DWORD    dwX;          //
    DWORD    dwY;          //窗口左上角坐标偏移量
    DWORD    dwXSize;
    DWORD    dwYSize;              //窗口的 X 向、Y 向尺寸
    DWORD    dwXCountChars;
    DWORD    dwYCountChars;   //控制台应用程序的 X，Y 向缓冲区大小
    DWORD    dwFillAttribute;    //设置窗口的文字和背景颜色
    DWORD    dwFlags;              // STARTUPINFO 结构中哪些成员有效
    WORD     wShowWindow;     //如何显示窗口
    WORD     cbReserved2;        //保留待用，必须为 0
    LPBYTE   lpReserved2;         //保留待用，必须为 NULL
    HANDLE   hStdInput;            //进程的标准输入句柄
    HANDLE   hStdOutput;          //进程的标准输出句柄
    HANDLE   hStdError;            //进程的标准出错句柄
} STARTUPINFO, *LPSTARTUPINFO；
```

dwFlags 标志着 STARTUPINFO 结构中哪些成员有效。可以有如表 9-3 所列选项。

表 9-3 dwFlags 可选标志

标记值	作用域
STARTF_USESHOWWINDOW	WShowWindow 有效
STARTF_USEPOSITION	DwX and dwY 有效
STARTF_USESIZE	DwXSize and dwYSize 有效
STARTF_USECOUNTCHARS	DwXCountChars and dwYCountChars 有效
STARTF_USEFILLATTRIBUTE	DwFillAttribute 有效
STARTF_USESTDHANDLES	HStdInput, hStdOutput, and hStdError 有效

lpProcessInformation 新创建的进程的信息。在调用 CreateProcess 后将返回此结构的指针，以得到所创建的进程的信息。此结构的内容如下：

```
typedef struct _PROCESS_INFORMATION { // pi
    HANDLE  hProcess;  //新创建的进程实例句柄
     HANDLE hThread; //新创建的线程实例句柄
    DWORD  dwProcessId; //新创建的进程 ID
    DWORD  dwThreadId; //新创建的线程 ID
     } PROCESS_INFORMATION;
```

2．进程当前目录问题

当应用程序进行文件操作的时候，应用程序就不得不与文件系统打交道。事实上，几乎所有的 Win32 应用程序都将操作文件 I/O 来获得持久性信息。这种数据的来源是不同的，有的来自于系统注册表(System Registry)，有的来自于数据库文件（DataSource)，有的来自于进程自用的数据文件。当进程操作自用文件的时候，为了稳定起见，一般都指定所操作的文件的文件系统全路径名。但是，有的应用程序却忽略了这一点，而只给出了相对路径。这将会导致某些应用程序找不到数据文件。此时，应用程序应该检查进程的当前目录，以确认当前目录是否正确。Win32 提供下列函数，来获取当前路径。

```
DWORD GetCurrentDirectory(
  DWORD  nBufferLength,   //  以字符为单位的记载当前目录的字符缓冲区长度
   LPTSTR    lpBuffer              //  记载当前目录的字符缓冲区地址);
```

这个函数将返回进程的当前目录，并保存在 lpBuffer 所指向的缓冲区内。

如果函数调用成功，GetCurrentDirectory 将返回路径缓冲区中所填入的字符串的长度。

另一个与之互补的一个函数：

```
BOOL SetCurrentDirectory(
  LPCTSTR  lpPathName     //  新路径字符缓冲区指针
);
```

当当前所设制的新路径存在时，返回非零值。

来看下面一段代码：

```
//…
if (FALSE==(WinExec(″c:\\somedir\\someapp.exe?,SW_SHOW)))
{
      MessageBox(?Error at execute the
            application?, ?System Error?,MB_ICONERROR);
      Return    …;
}
```

在 someapp.exe 中，存在一下代码：

```
FILE * fp=fopen(?_data.dat?,?r?);
if (NULL==fp)
{
      MessageBox(?Error at open data file?, ?File System Error?,MB_ICONERROR);
```

```
        Return    …;
}
```

更有的程序根本不检查文件句柄，而是直接访问数据：

```
TCHAR buf[100];
int num=0;
fscanf(?%d%s?,&num,buf);
```

前一段代码本身并无严重错误，只是当 someapp.exe 需要加载数据文件时，将可能引起"文件无法访问"的错误，或者将引起文件 I/O 错误，令进程不得不终止。如果在进行文件操作之前，先设置好当前目录，则不会产生这种情况。例如：

```
TCHAR CurPath[MAX_PATH];
GetCurrentDirectory(MAX_PATH,CurPath);
//…
if (FALSE==(WinExec(?c:\\somedir\\someapp.exe?,SW_SHOW)))
{
    MessageBox(?Error at execute the application?,
        ?System Error?,MB_ICONERROR);
    Return    …;
}
if (FALSE==SetCurrentDirectory(CurPath))
{
    MessageBox("Error at reset current directory of this process",
                "File  system  Error",
                MB_ICONERROR);
    return ;
}
```

顺便说一下，传给 GetCurrentDirectory 的字符串缓冲区长度应尽可能大一些，因为你的应用程序所处的文件系统可能是 FAT、FAT32 或 NTFS。Visual C++定义了一个宏，即 MAX_PATH 用以解决这一问题。

3．进程创建示例

可以按如下方法创建一个进程：

```
STARTUPINFO stif;
stif.cb=sizeof(stif);
stif.dwFlags|=STARTF_USESHOWWINDOW|STARTF_USEPOSITION;
stif.wShowWindow=SW_SHOW;
stif.dwX=200;
stif.dwY=200;
stif.lpTitle="sample process create";
```

```
PROCESS_INFORMATION prif;
ZeroMemory(&stif,sizeof(prif));
BOOL bRtnCreate=FALSE;
bRtnCreate=    CreateProcess("Regedit.exe","Regedit",
NULL,NULL,FALSE,CREATE_UNICODE_ENVIRONMENT,
NULL,NULL,&stif,&prif);
if (bRtnCreate==FALSE)
      {
            MessageBox("Error at create the process",
                        "System  Error",
                        MB_ICONINFORMATION);
            return;
      }
```

9.2.4　终止进程

当你的应用程序不再需要某个进程时，就需要结束进程，有 3 种方法可以终止进程：

1．ExitProcess 函数

调用该函数将结束当前进程，对其他进程毫无影响。

函数原型为：

```
VOID ExitProcess(   UINT uExitCode    // exit code for all threads);
```

正如前面所说的，此函数终止当前进程本身。进程结束以后，其所占用的地址空间中的资源将被释放，该进程的所有子线程将被终止。但由此进程所创建的子进程不会终止，而将仍然占据系统所分配的地址空间及其资源。值得注意的是，进程中的每一个线程调用此函数，都会引起进程的终止。

获得退出码有两种方法：

- GetExitCodeProcess　获得进程的退出码。
- GetExitCodeThread　获得线程的退出码。

```
BOOL GetExitCodeProcess(   HANDLE hProcess, //  进程句柄
                              LPDWORD  lpExitCode     //  接受返回的退出值的内存地址);
BOOL GetExitCodeThread(
          HANDLE hThread,           //  线程句柄
          LPDWORD  lpExitCode      //  接受返回的退出值的内存地址);
```

2．TerminateProcess 函数

调用该函数不仅可以终止当前进程本身，而且可以终止别的进程。

函数原型为：

```
BOOL TerminateProcess(   HANDLE hProcess, //  要退出的进程句柄
```


此为试读，需要完整PDF请访问：www.ertongbook.com

```
UINT uExitCode    // 进程的退出值);
```

此函数可终止 hProcess 所指出的进程。在 Windows NT 与 Windows 2000 中，在系统状态栏上右击鼠标，启动任务管理器，在进程区即可看到系统的所有进程。可以选择"结束任务"来结束进程。

3．通过 Wait 函数实现

我们在开发时经常需要在启动一个进程之后等待其结束后再继续运行。在这里提供了一个名为 Wait 的函数，它会为你完成上面的功能。

实现的思想是在启动进程后等待其结束，由于进程是一种资源，而资源的句柄在 Win32 中可以作为核心量使用，你可以使用 WaitForSingleObject 等待核心量状态改变为有信号状态，对进程来讲当进程结束时其状态转变为有信号。在本例中使用了一个单独的线程来启动进程并等待结束。

源代码如下：

```
Wait()
BOOL Wait(CString szCmdLine)
{
    LPTHREADINFO pThreadInfo = new THREADINFO;
    CEvent *pThreadEvent = new CEvent(FALSE, TRUE);
    ASSERT_VALID(pThreadEvent);
    if(pThreadInfo)
    {
        pThreadInfo->pTermThreadEvent = pThreadEvent;
        pThreadInfo->strPathName = szCmdLine;
        AfxBeginThread(LaunchAndWait, pThreadInfo);
        WaitForSingleObject(pThreadEvent->m_hObject, INFINITE);
        return TRUE;
    }
    return FALSE;
}
LaunchAndWait()
UINT LaunchAndWait(LPVOID pParam)
{
    LPTHREADINFO pThreadInfo = (LPTHREADINFO) pParam;
    PROCESS_INFORMATION stProcessInfo;
    if(LaunchApplication(pThreadInfo->strPathName, &stProcessInfo))
    {
        HANDLE hThreads[2];
        hThreads[0] = pThreadInfo->pTermThreadEvent->m_hObject;
```

```
            hThreads[1] = stProcessInfo.hProcess;
           DWORD dwIndex = WaitForMultipleObjects(2, hThreads, FALSE, INFINITE);
            CloseHandle(stProcessInfo.hThread);
            CloseHandle(stProcessInfo.hProcess);
            pThreadInfo->pTermThreadEvent->SetEvent();
            if(pThreadInfo)
                 delete  pThreadInfo;
      }
      else
            pThreadInfo->pTermThreadEvent->SetEvent();
     return  0;
}
LaunchApplication()
BOOL LaunchApplication(LPCTSTR pCmdLine, PROCESS_INFORMATION
*pProcessInfo)
{
     STARTUPINFO  stStartUpInfo;
     memset(&stStartUpInfo, 0, sizeof(STARTUPINFO));
     stStartUpInfo.cb = sizeof(STARTUPINFO);
     stStartUpInfo.dwFlags = STARTF_USESHOWWINDOW;
     stStartUpInfo.wShowWindow = SW_SHOWDEFAULT;
      return CreateProcess(NULL, (LPTSTR)pCmdLine, NULL, NULL, FALSE,
           NORMAL_PRIORITY_CLASS, NULL,
          NULL, &stStartUpInfo, pProcessInfo);
}
```

9.2.5　进程间的通信方式

在 Windows 2000 中，为实现进程间平等的数据交换，用户可以有如下几种选择：

- 使用内存映射文件。
- 通过共享内存 DLL 共享内存。
- 向另一进程发送 WM_COPYDATA 消息。
- 调用 ReadProcessMemory 以及 WriteProcessMemory 函数，用户可以发送由 GlobalLock(GMEM_SHARE,...)函数调用提取的句柄、GlobalLock 函数返回的指针以及 VirtualAlloc 函数返回的指针。

1．利用内存映射文件实现 Win32 进程间的通信

Windows 2000 中的内存映射文件机制允许我们在 Win32 进程中保留一段内存区域，把目标文件映射到这段虚拟内存中。在程序实现中必须考虑各进程之间的同步。具体实现步

骤如下：

首先我们在发送数据的进程中需要通过调用内存映射 API 函数 CreateFileMapping 创建一个有名的共享内存：

```
HANDLE CreateFileMapping(
HANDLE  hFile,       //  映射文件的句柄，
//设为 0xFFFFFFFF 以创建一个进程间共享的对象
LPSECURITY_ATTRIBUTES lpFileMappingAttributes,       //  安全属性
DWORD  flProtect,       //  保护方式
DWORD  dwMaximumSizeHigh,       //对象的大小
DWORD  dwMaximumSizeLow,
LPCTSTR  lpName       //  必须为映射文件命名
);
```

与虚拟内存类似，保护方式可以是 PAGE_READONLY 或是 PAGE_READWRITE。如果多进程都对同一共享内存进行写访问，则必须保持相互间同步。映射文件还可以指定 PAGE_WRITECOPY 标志，可以保证其原始数据不会遭到破坏，同时允许其他进程在必要时自由操作数据的拷贝。在创建文件映射对象后可以调用 MapViewOfFile 函数映射到本进程的地址空间内。

下面说明创建一个名为 MySharedMem 的长度为 4096 字节的有名映射文件：

```
HANDLE hMySharedMapFile=CreateFileMapping((HANDLE)0xFFFFFFFF)，
NULL，PAGE_READWRITE，0，0x1000，"MySharedMem")；
```

并映射缓存区视图：

```
LPSTR pszMySharedMapView=(LPSTR)MapViewOfFile(hMySharedMapFile，
FILE_MAP_READ|FILE_MAP_WRITE，0，0，0);
```

其他进程访问共享对象，需要获得对象名并调用 OpenFileMapping 函数。

```
HANDLE hMySharedMapFile=OpenFileMapping(FILE_MAP_WRITE，
FALSE，"MySharedMem");
```

一旦其他进程获得映射对象的句柄，可以像创建进程那样调用 MapViewOfFile 函数来映射对象视图。用户可以使用该对象视图来进行数据读写操作，以达到数据通信的目的。

当用户进程结束使用共享内存后，调用 UnmapViewOfFile 函数以取消其地址空间内的视图：

```
if (!UnmapViewOfFile(pszMySharedMapView))
{ AfxMessageBox("could not unmap view of file"); }
```

2．利用共享内存 DLL

共享数据 DLL 允许进程以类似于 Windows 3.1 DLL 共享数据的方式读写数据，多个进程都可以对该共享数据 DLL 进行数据操作，达到共享数据的目的。在 Win32 中为建立共享内存，必须执行以下步骤：

首先创建一个有名的数据区。这在 Visual C++中是使用 data_seg pragma 宏，使用 data_seg

pragma 宏必须注意数据的初始化：

```
#pragma data_seg("MYSEC")
char MySharedData[4096]={0};
#pragma data_seg()
```

然后在用户的 DEF 文件中为有名的数据区设定共享属性。

```
LIBRARY TEST
DATA READ WRITE
SECTIONS
.MYSEC READ WRITE SHARED
```

这样每个附属 DLL 的进程都将接受到属于自己的数据拷贝，一个进程的数据变化并不会反映到其他进程的数据中。

在 DEF 文件中适当地输出数据。以下的 DEF 文件项说明了如何以常数变量的形式输出 MySharedData。

```
EXPORTS
MySharedData CONSTANT
```

最后在应用程序（进程）按外部变量引用共享数据。

```
extern _export"C"{char * MySharedData[];}
```

进程中使用该变量应注意间接引用。

```
m_pStatic=(CEdit*)GetDlgItem(IDC_SHARED);
m_pStatic ->GetLine(0,*MySharedData,80);
```

3．用于传输只读数据的 WM_COPYDATA

传输只读数据可以使用 Win32 中的 WM_COPYDATA 消息。该消息的主要目的是允许在进程间传递只读数据。Windows 95 在通过 WM_COPYDATA 消息传递期间，不提供继承同步方式。SDK 文档推荐用户使用 SendMessage 函数，接受方在数据拷贝完成前不返回，这样发送方就不可能删除和修改数据：

```
SendMessage(hwnd,WM_COPYDATA,wParam,lParam);
```

其中 wParam 设置为包含数据的窗口的句柄。lParam 指向一个 COPYDATASTRUCT 的结构：

```
typedef struct tagCOPYDATASTRUCT{
    DWORD  dwData;//用户定义数据
    DWORD  cbData;//数据大小
    PVOID  lpData;//指向数据的指针
}COPYDATASTRUCT;
```

该结构用来定义用户数据。

4．直接调用 ReadProcessMemory 和 WriteProcessMemory 函数实现进程间通信

通过调用 ReadProcessMemory 以及 WriteProcessMemory 函数，用户可以按类似于 9.3.1 小节的方法实现进程间通信。

在发送进程中分配一块内存存放数据，可以调用 GlobalAlloc 或者 VirtualAlloc 函数实现：

```
pApp->m_hGlobalHandle=GlobalAlloc(GMEM_SHARE,1024);
```

可以得到指针地址：

```
pApp->mpszGlobalHandlePtr=(LPSTR)GlobalLock
(pApp->m_hGlobalHandle);
```

在接收进程中要用到用户希望影响的进程的打开句柄。为了读写另一进程，应按如下方式调用 OpenProcess 函数：

```
HANDLE hTargetProcess=OpenProcess(
STANDARD_RIGHTS_REQUIRED|
PROCESS_VM_REDA|
PROCESS_VM_WRITE|
PROCESS_VM_OPERATION,//访问权限
FALSE,//继承关系
dwProcessID);//进程 ID
```

为保证 OpenProcess 函数调用成功，用户所影响的进程必须由上述标志创建。

一旦用户获得一个进程的有效句柄，就可以调用 ReadProcessMemory 函数读取该进程的内存：

```
BOOL ReadProcessMemory(
HANDLE  hProcess,       //  进程指针
LPCVOID  lpBaseAddress,       //  数据块的首地址
LPVOID  lpBuffer,       //  读取数据所需缓冲区
DWORD  cbRead,       //  要读取的字节数
LPDWORD  lpNumberOfBytesRead
);
```

使用同样的句柄也可以写入该进程的内存：

```
BOOL WriteProcessMemory(
HANDLE  hProcess,       //  进程指针
LPVOID  lpBaseAddress,       //  要写入的首地址
LPVOID  lpBuffer,       //  缓冲区地址
DWORD  cbWrite,       //  要写的字节数
LPDWORD lpNumberOfBytesWritten
);
```

如下所示源代码演示了如何读写另一进程的共享内存中的数据：

```
ReadProcessMemory((HANDLE)hTargetProcess,
(LPSTR)lpsz,m_strGlobal.GetBuffer(_MAX_FIELD),
_MAX_FIELD,&cb);
WriteProcessMemory((HANDLE)hTargetProcess,
```

(LPSTR)lpsz,(LPSTR)STARS,
m_strGlobal.GetLength(),&cb);

9.3 线程

9.3.1　线程概述

线程是一个能独立于程序的其他部分运行的作业。线程属于一个进程，并可获得自己的 CPU 时间片。基于 Win32 的应用程序可以使用多个可执行的线程，称为多线程。依靠生成多个线程，应用程序能够完成一些后台操作，例如计算，这样程序就能运行得更快。当线程运行时，用户仍能继续影响程序。

在 Win32 中，进程是没有活力的，它只占用 4GB 的虚拟地址空间，而不做任何事情，所有的处理都在线程中进行。Win32 平台支持多线程编程，使得应用程序可以用不同的线程处理不同的任务，这在多处理器机器上显得十分重要。

每个进程都有一个主线程，但可以建立另外的线程。进程中的线程是并行执行的，每个线程占用 CPU 的时间由系统来划分。可以把线程看成是操作系统分配 CPU 时间的基本实体。系统不停地在各个线程之间切换，它对线程的中断是汇编语言级的。系统为每一个线程分配一个 CPU 时间片，某个线程只有在分配的时间片内才有对 CPU 的控制权。实际上，在 PC 机中，同一时间只有一个线程在运行。由于系统为每个线程划分的时间片很小（20ms 左右），所以看上去好像是多个线程在同时运行。

进程中的所有线程共享进程的虚拟地址空间，这意味着所有线程都可以访问进程的全局变量和资源。这一方面为编程带来了方便，但另一方面也容易造成冲突。虽然在进程中进行费时的工作不会导致系统的挂起，但这会导致进程本身的挂起。所以，如果进程既要进行长期的工作，又要响应用户的输入，那么它可以启动一个线程来专门负责费时的工作，而主线程仍然可以与用户进行交互。

9.3.2　创建线程

Win32 中线程分用户界面线程（User_interface Thread）和工作者线程（Worker Thread）两种。用户界面线程拥有自己的消息泵来处理界面消息，可以与用户进行交互。工作者线程没有消息泵，一般用来完成后台工作，而不处理用户的输入及维护图形界面。

1．MFC 应用程序中的线程创建

MFC 应用程序的线程对象由 CWinThread 表示。在多数情况下，程序不需要自己创建 CWinThread 对象。调用 AfxBeginThread 函数时会自动创建一个 CWinThread 对象。

AfxBeginThread 函数的声明为：

CWinThread* AfxBeginThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0, DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

此为试读，需要完整PDF请访问：www.ertongbook.com